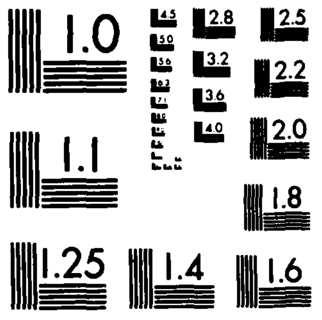
1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

ROYAL SIGNALS AND RADAR ESTABLISHMENT

REPORT 82010

Title: THE DISCUS HARDWARE SYSTEM

Author: H.S. Field-Richards

Date: July 1982

## SUMMARY

This report describes the hardware philosophy behind the Distributed Control microprocessor System (DISCUS) developed at RSRE. DISCUS is an array of up to 15 asynchronous microprocessors that are star connected around a global store. The system is compared to other architectures, and the general philosophy behind the design of DISCUS. This is to keep the hardware and software as simple as possible while achieving the aims of system recovery and diagnostics. Enhancements for the future such as the choice of microprocessor and bus are discussed as well as the present and possible future recovery aids.

discus

"That every boy and every gal,
  That's born into the world alive,
Is either a little Liberal,
  Or else a little Conservative."

Iolanthe (Act II Scene 1)        W.S. Gilbert

(The relevance of this rather obscure quote can be found in the report on the operating system - reference [8])

## CONTENTS

## LIST OF ILLUSTRATIONS etc

# 1. INTRODUCTION

In the past 40 years, there has been a continued improvement in the quality and size of the basic building blocks that are used to make computers. In the early days these basic building blocks available to the designer were the normal components of the day: resistors, capacitors and valves. The modules they were used to make, such as bistables, could be several cubic inches in size. Over the years the components continued to be reduced in size with the advent of the transistor and then that of small scale integration. The basic building blocks became complete sub-assemblies that consumed very little power and space in relation to their ancestors. The result was an increase in reliability by several orders of magnitude, and the proliferation of the computer became more widespread. It is during the last decade that there has been the most dramatic change in the systems within the computer. Large scale integration, with the ability to produce such devices as the 64 kbit memory and the microprocessor, will be the key area in the future development of all levels of computing science. With the cost of these components being relatively trivial, it is possible to reconsider the way in which we make computers. The modern building blocks have become the complete sections of the computer: CPU, store etc. In some cases these blocks can be a complete processor so that the designer can use the computer as an available design component in its own right.

It was how a recoverable computer might best be designed without the restraint of the hardware expense that prompted the present research. We felt that the advantages from an array of processors carrying out what a single processor normally does would be considerable. In particular we wished to explore areas that this general concept would give us -

1. Improve the containment of errors.

2. Reduce the software complexity if possible.

3. Make redundant hardware.

Not all these aims were achieved, or even attempted, and some other advantages were accrued as the work progressed.

The final machine is an array of asynchronous microprocessors that we have applied to a particular real-time application (in our case a small telephone exchange). There can be up to 15 (local) processors that can be used in a star configuration around a central (global) store. It is designed around a bus that allows a variety of microprocessors to be utilised in the array. Since the peripheral cards (memory, I/O etc) are communicating to this bus rather than a particular microprocessor, these cards should be constant for a whole range of microprocessors.

The general philosophy behind the design of DISCUS has been to keep the hardware and software as simple as possible while achieving the aims of system recovery and diagnostics. This was achieved by the maximum use of "intelligent" peripheral cards, that allows partitioning of the software into functional blocks.

DISCUS has no form of hardware synchronisation between the processors. The only synchronisation is at an operating system level. Each processor is able to make a claim for the global store at any time, independently of any other processor and totally unsynchronised with them. Also, all the activity on each local DISCUS bus is carried asynchronously. This includes the direct memory accessing of the intelligent peripheral cards to the local DISCUS store. Dynamic store is asynchronously refreshed in both the local and global stores. Within a full DISCUS multiprocessor there are at least three or four levels of asynchronous working. I have developed a circuit which I believe makes negligible the problems of asynchronous working in DISCUS. I do not claim that the problem of the "meta-stable" latch, [9][10][15], has been cured, but that the probability of it happening has been reduced to an insignificant level.

A further innovation has been to assign one processor for every application's function. This means that the scheduler in the operating system can be dispensed with, and that there is no form of dynamic reallocation of the processors during the running life of the DISCUS. Each function is assigned when the applications program is designed. The hardware can be loaded with any function since the communications between the processors is assigned in the software. There is no overall control processor except when the system is initially loaded; when the applications programs start running they do so autonomously. It is possible to load the same function onto two or more processors and have them share the work. This helps to relieve potential bottlenecks in the system without the need for reworking the functional split of the applications program.

Great emphasis is placed on how the system protects itself from deliberate or accidental corruption of system data by the user. Address overlays between the local and global store are used as well as numerous checks on data in the system in both hardware and software.

Section 2 of this report gives an introduction to multiprocessing in general and why we should want to take the DISCUS approach. Section 3 describes the present hardware system, with section 4 discussing our preliminary approaches to recovery within DISCUS, particularly with the hardware in mind. Sections 5 and 6 look at the microprocessors and busses respectively that might be used in a DISCUS like machine in future. Section 7 sums up what has been done with a brief re-cap of what should be done in future. Bear in mind that this report should be read with "The DISCUS Hardware Description and Appendices" memo [29], and "The DISCUS Operating System" report [8] close by.

## 2. MULTI-PROCESSING AND DISCUS

The design and use of any computer system must be considered over a number of years. As time progresses it becomes inevitable that there will be changes in the basic function of the system and also a change due to curing faults that must occur. The people that carry out the system enhancements and the repairs are not the same. As well as this the calibre of the staff can decline in capability over this period. For instance, consider the case of a compiler or operating system that needs to be written for a particular computer. A very high calibre team are employed by the company to do the initial design and the immediate debugging that is required over the first few man-years of use (in practice this period could be only a few months). Assuming that the company employing them does not provide considerable incentive for them to stay in terms of either money or interest in work, these people will move on to where the jobs or incentives are more alluring. The company who market the compiler are now obliged to maintain and support their product. In order to maintain their product the company will find it necessary either to employ more software engineers or to use the original team on an external basis. It is more likely that the former will be the solution since it is going to be more under control of the host company. The people that will be attracted, by what is only a maintenance job, are unlikely to be of the same quality as the original team. As the complexity increases and the current software team becomes more and more distanced from the original one, there must come a point when the system becomes unmanageable. This manifests itself in a continued degradation of the service and an inability of the system to protect itself from unforeseen random events. This is because bugs creep into the system owing to an incomplete understanding of a large software system.

It is thus very important that the system is resistant to errors. In order to achieve this a considerable amount of discipline must be used in the design of the system. Up to a few years ago this was achieved in the single processor computer by having large operating systems that carried considerable overheads in checking the integrity of the operations being done. Although the applications programs may have been well protected they relied entirely on the discipline of the programmer and the reliability of the operating system.

We considered that a new look could be taken at the problems inherent in large amounts of software. This decision was made at a time when the microprocessor was becoming available cheaply, and thus hardware was no longer a restraining influence in the construction of a computer.

As I mentioned above, one of the keys to producing a reliable system was discipline. What we felt necessary was a way of forcing the user to use this discipline, so that he was unable to structure his programs in any other way than that we set him. Another point is that if he does make errors (or indeed the operating system makes them) they should be contained so that the effect on other parts of the system is minimal.

### 2.1 THE FUNCTIONAL APPROACH

Any non-trivial task that is undertaken is split, consciously or unconsciously, into a set of smaller sub-tasks or functions. These functions are further split down into smaller functions until it is inconvenient or impractical to go any further. With a computing problem these functions are generally program segments or pieces of hardware. The program segments are able to run a process to carry out the job required. Each piece of functional software operates on one job, or transaction, at a time and the job is passed from function to function.

It is convenient to break down a task into functions of manageable size. This process is known as "functional decomposition". Before going on the phrase "manageable size" is worth considering. I consider that it implies the following attributes.

i. It must be easily understood in its entirety by one man.

ii. The documentation must be easy to follow and must parallel the structure of the software.

Basically a function should be a "brain sized" package.

On a single processor computer processes running on these functions are dynamically created and destroyed and run by the operating system scheduler. Each is given a certain amount of time on the processor. The time is terminated by a logical wait point in the program, such as waiting for an unavailable resource (known as co-operative scheduling), or a "watch-dog" timer (known as competitive or pre-emptive scheduling). Each time the operating system must correctly save away the context. Then it must l[illegible] the context of the next process to be run (usually determined by some [illegible]m of priority list) which is then run for its alloted time. There a[illegible] many disadvantages to this scheme.

i. It is impossible to improve the performance of a badly split [illegible]m except by redesigning it - a considerable task usually.

ii. The operating system is a very large complex piece of code that includes a scheduler, which implies unreliability. If any form of automatic procedure was going to be tried to prove this very basic part of the system correct, it would be better to have the operating system as small as possible.

iii. An error in one part of the system can affect the rest of the system.

The advantages are:

i. The peripherals are connected to all the functions.

ii. Code which is common to other functions can be shared. For instance only one copy is needed of the operating system.

iii. When data is sent from one function to another the actual data need not be passed, only pointers to where that data may be found.

To sum up then, it is convenient to break down the required overall task into functions of manageable size. These functions can then be mapped onto the particular computer to be used. However the design is done, the end result will be a system consisting of several functions, which may or may not be dynamically created and destroyed and which are mapped onto one or more physical processors. Usually this involves the provision of a complicated operating system to provide scheduling and time sharing of the functions. With the advent of the microprocessor, it has become becomes feasible to consider providing one processor per fixed (non-dynamic) process. It is this idea which we took for DISCUS.

## 2.2 THE DISCUS APPROACH

The required software system is split into functions and each function is executed on a self-contained microcomputer, and that computer executes only that function. Thus no complicated function changing, or scheduling operating system is required. Error containment between functions is achieved at the first level by their actual physical and electrical separation.

I feel that the idea of one processor per process is easy to understand. No complicated operating system and/or master processor is required. In the mature system, programs will reside in PROM memory, minimising errors. The inter-function communicators are simple and precisely defined.

The system is also flexible, because processing power is available in small processing steps. The design of DISCUS makes it possible to add a processor, complete with PROM program, either as a duplicate for an existing function, nominally to double the throughput of that function, or as a replacement for a failed processor. I will return to exactly how the DISCUS array is connected in section 3.

## 2.3 MULTI-PROCESSOR SYSTEMS

To say that a system is a multi-processor is not to tell the complete story. There are several classifications which I must define before any further discussion is continued. The following list is based on work presented by Searle and Freberg [2], in a general discussion of multi-computers.

**REDUNDANT SYSTEMS** provide unit replacement on either an automatic or manual basis. Usually in modern systems the former is employed, which requires sophisticated failure detection and automatic systems recovery. The problems *of recognising a failure when it does occur is difficult. At present* detecting all the faults in a system is virtually impossible, although detection to a degree where the system is able to give almost perfect service is now relatively common. These systems are most prevalent in areas where continued correct service is paramount: areas such as spacecraft systems and the nuclear industry. This area has been very well explored by many and I do not intend to consider it for DISCUS since the latter is not intended to be primarily a redundant system.

**PIPELINING** is a way of exploiting the basic internal operations of a computer. The normal "fetch-do" cycle can be interleaved by more than one processor on the same bus, hopefully, therefore, achieving higher data throughput. The new breed of micros (8086 etc) use a form of look-ahead pipelining to achieve higher processor speeds. A first-in first-out instruction store on the code input is used to buffer instructions up to 6 ahead. The Intel 8086 is in some respects 2 tightly coupled micros co-operating together to form a high performance 16 bit micro. The 2 parts are an Execution Unit which performs the basic processing functions, containing the data registers and the arithmetic unit, and a Bus Interface Unit that provides the instruction fetching and bus control, allowing best use of the memory and peripherals. The pipelining I am considering here is generally used at a fairly low-level in the structure of the computer and need not concern us any further here.

**NETWORK SYSTEMS** are perhaps the most familiar form of interconnected computer systems to most people. The connections between them are via long distance data channels which must in most cases be serial. Because of this serial working, rigid protocols must be used such as HDLC. These networks are essentially data links rather than problem-sharing networks. Again I do not intend to go any further with them.

**MULTI-PROCESSOR SYSTEMS** consist of a set of tightly coupled processors working co-operatively to perform a single task. Within this classification we can identify two further sub-divisions given below:

i. The first type is a little difficult to name: Searle and Freberg call it a "**Multi-processor**" System, but this is perhaps rather too vague and it might be better to add the further classification of "**transactional-based multi-processor**" as opposed to the "**functional-based**" system below. In the transactional system there is only a single executive/scheduler which can dynamically allocate tasks to each processor. There might be only one cpu in which case the system should really be called a "multi-process" system.

ii. The other classification is the "**Functional Based System**" or as it is sometimes known the "**Distributed computer system**". The hardware is similar to that above, but here, instead of each processor being capable of running any of the functions, each unit performs a dedicated function and no re-allocation of this function takes place during the running life of the machine. It is only now that the hardware has become available in sufficiently small form and cheaply enough to warrant this approach to be tried - DISCUS is such a system.

It is now assumed that most new developments in technology are done because they offer a benefit over their ancestors, although this benefit is often not immediately apparent. It is instructive, therefore, to consider the advantages that we hope to derive from connecting more than one computer together in the ways indicated above.

The speed of system response obviously must be among the prime reason for multi-processing. Despite the advent of microprocessors and their short internal connecting length within the integrated circuit, the time is rapidly approaching where the speed of light is the limiting factor in pure speed performance of a single processor.

I must emphasise that I do not consider that absolute processor speed is an essential criteria for computing power, and that speed combined with a measure of the amount of parallelism must offer a far greater indication within a system of that system's power and throughput. This parallelism can be obtained by several methods: multiple processors, increasing the word length to name but two.

By causing the system to have an amount of parallel processing, we can achieve an apparent increase in computer speed. How much increase we actually get is not as simple as doubling the speed for doubling of the number of processors, and the relation between these two quantities is not linear. The reasons for this lies in the hardware constraints such as store contention, and the software deficiencies imposed on the original selection of functions to be made parallel. In future the penalty of having designed a badly split system may not be so concerning, since hardware is cheaper than software and adding another processor to help a badly split system would be cheaper than redesigning it.

Resilience of the system to random and deliberate events that cause the system to fail must also be a very strong reason. The ability of a machine to reconfigure itself by function sharing when a processor fails is an important one in the consideration of multi-computers, especially in real time systems. The rate of system degradation in a multi-processor system can be arranged to be gradual in comparison to its single processor brethren which usually fail catastrophically. Methods of failure detection and the recovery to be taken in the event of this failure must concern greatly the designers of real time multi-processors.

## 2.4 TYPES OF PROCESSOR INTERCONNECTION

The way in which computers are connected to form a multi-computer system has a fundamental effect on the final system performance both in terms of the speed and reliability. How these various connections are derived do not always have a firm foundation in science but result from such factors as the job the computer is going to perform. Often it can reflect the designer's own personal preferences which, in the past has led to a less than optimum service from the system. It is becoming possible however, with modern, cheap hardware to optimise this design over 2 or 3 iterations to get the best performance from the system. Although I must add that iteration is no substitute for sound design in the early stages.

Up to now there have been a number of classes of computer interconnection. I can summarise these briefly as [2][1]:

i. Dedicated Bus
ii. Multiporting
iii. Single Time Shared Bus
iv. Multiple Time Shared Bus
v. Crossbar
vi. Star Connected
vii. Ring

Obviously these cannot be completely watertight divisions as many multi-processor system are amalgams of some of the various types. Also the classifications given here cannot be considered as definitive but are presented only as a guide.

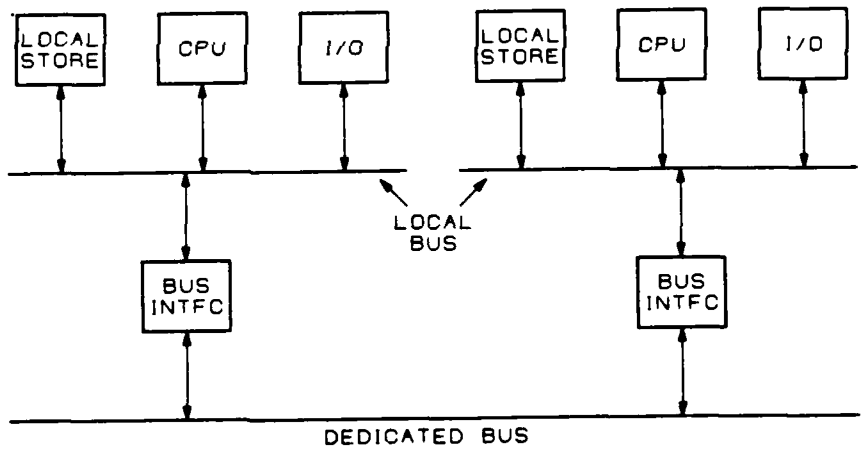

**FIG 1. Dedicated Bus System**

### 2.4.1 Dedicated Bus

A **Dedicated Bus System**, Fig 1, is one in which there is no permanent link directly between processors. There is no intermediate store between the 2 to provide common data base or interprocess/function channels. Any data base that is required in the system must either be held common in all processors or else in only one of the processors dedicated to data base handling. Each processor must obviously be capable of accessing neighbours' stores in order to pass messages between themselves. It follows therefore, that some form of store protection local to each processor is required to prevent accidental or deliberate corruption of the local stores within the system. Software and hardware methods can be used. The software protection is afforded by having all interprocessor accesses carried out via an operating system that must be resident within each processor. The hardware protection can be done by having only a certain area of each store reserved for "global" use and thus making it physically impossible for one processor to access another's local program or workspace. It should be noted that the system-generator should handle all the memory allocation at compile time for each of the segments in the processors, with some of the access violations being detected here.

One modern system which could be thought of as a dedicated bus system is one formed from a number of Intel SBC80/30s [4]. Each processor is able to access all its local store as well as that of everyone else. Each CPU, memory and I/O resides in one card with the edge connection providing access to the dedicated bus. By making this bus their own standard Multibus microprocessor bus, it is possible for Intel to put on this bus common resources such as store, I/O, real time interrupts etc. However, by doing this it must cross a classification boundary and borrow from the star connected class.

An advantage of the dedicated bus approach is obviously one of simplicity and thus low cost, especially in cases such as Intel's. If however the dedicated bus is distributed too much geographically, then there will arise complications in providing suitable electrical termination and driving to ensure best use of the bus. The maximum speed will obviously suffer, as will the cost of providing more hardware. It is best to use very tightly coupled systems for this approach to prevent undue length of the dedicated bus. For reliability, it is necessary to provide bus replication (Multiporting see below), and it then becomes increasingly more difficult to provide easy connections within the system because of the amount of bus interconnections required to each processor.

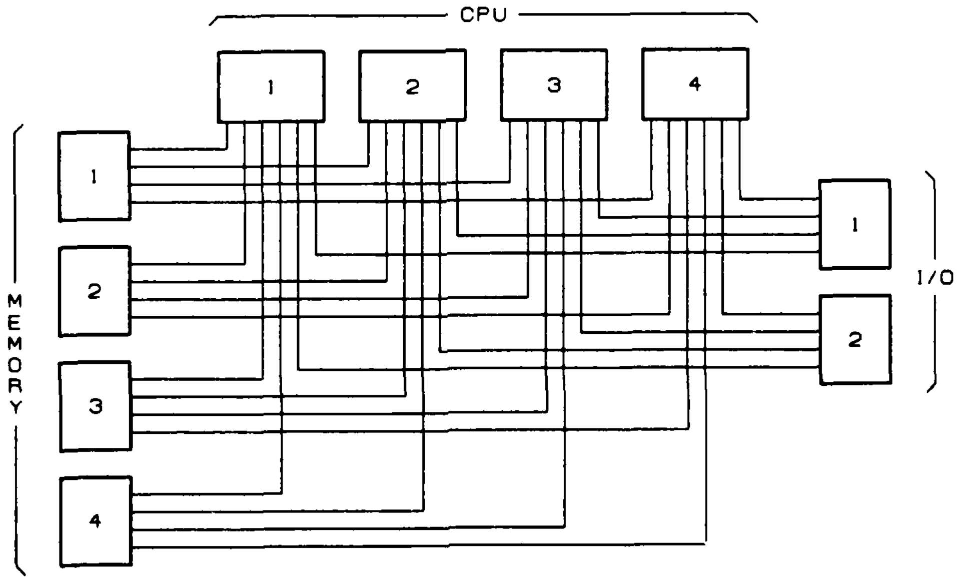

FIG 2. **Full Multiport System**

### 2.4.2 Multiport Systems

**Multiport Systems** follow on from trying to make a dedicated bus system more reliable by providing more interconnection paths between the various elements of the system. There are 2 variants of the system: the one with all the stores, CPUs, I/O devices etc being interconnected fully with no common (global) store, and the other with local stores and several dedicated busses.

The fully interconnected multiport system, Fig 2, provides better throughput since the contention for individual stores must be lower than those systems with single connections. Each memory, etc must have an interface handling the inputs from each CPU and arbiting between the various store requests. The arbiter is of necessity complex from sheer quantity of inputs and thus does not lend itself to system expansion. The size of the final system and possible expansions must be known clearly at the start; which is not always practical or possible. The cost and design of the connectors and the cable represents a substantial part of the total cost of the system. In general this type of system is not found in small mini or micro applications because of its complexity.

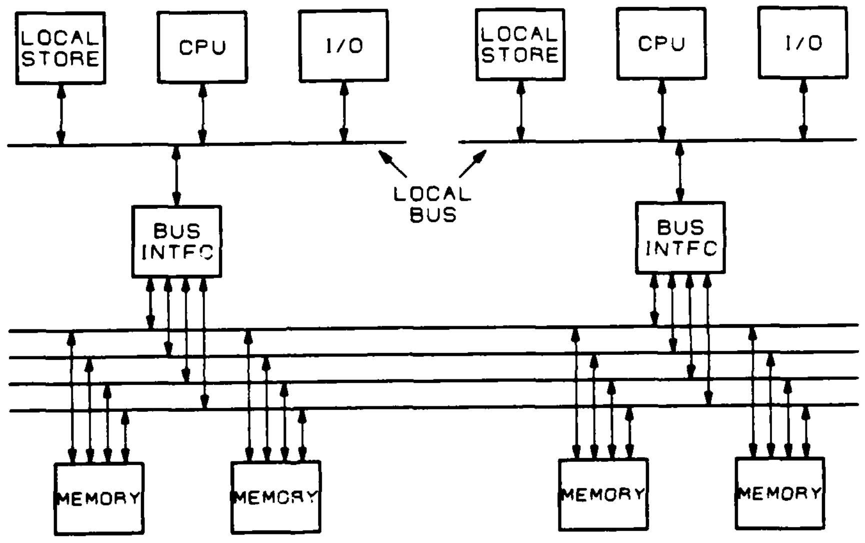

**FIG 3.** **Multiport System with Local Store**

The other type, Fig 3, is one that could be fairly easily applied to these lower-end processors and is really only an enhanced version of the distributed bus systems discussed above. There are now several of these dedicated data paths so that contention is reduced and there is also bus redundancy by having more than one bus. The system only uses those busses for data or message passing rather than for the program, the latter carried out on the local bus with its own local store. It is still necessary to be fairly careful in the initial system design because of the complex nature of the bus interfaces and controllers.

### 2.4.3 Single Time Shared Bus

It is possible to consider this bus, Fig 4, as a resource similar to the various memory and peripheral devices that must be requested when use is required of it. Because of this there must be some form of contention resolving circuit. The amount of contention must depend on several factors.

The first is the speed of a processor and the time it actually has control of the bus; it might be possible to have a machine with a relatively slow instruction execute time but a very fast instruction fetch. This would occur in a processor where the instructions are of a far higher level than the micro-instruction code within the processor. The memory cycle must also contribute to the bus occupancy by a processor. If a refresh cycle on dynamic semiconductor store is taking place, an individual processor requiring data from that store would obviously be delayed. This particular problem can be helped by designing a suitable refresh mechanism which interleaves refreshing with processor requests. A final factor in overall system throughput is the number of master control devices using the bus. There must come a time when the number of processors is such that the addition of another processor can cause the system performance to degrade rather than improve.

For all this, the single time shared bus does have its advantages. The cost of the module interconnections are minimal and it is possible to add new devices onto the bus very easily. These factors are offset by there being no redundancy in the system, a failure of the bus will result in total system loss. If any sort of redundancy is required, it is necessary to go to a multiple time shared bus approach.

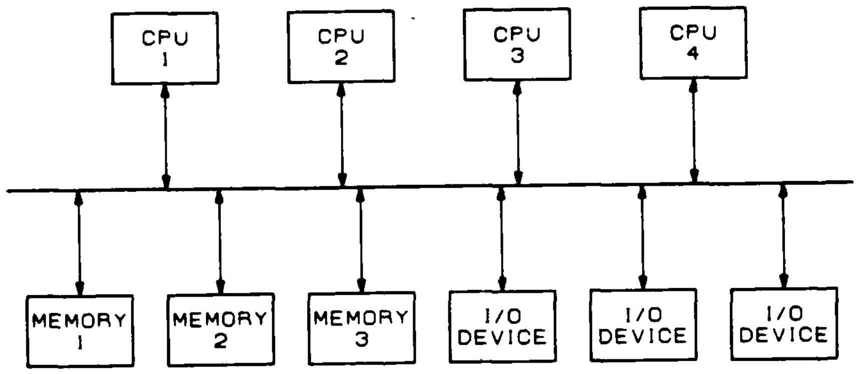

**FIG 4. Single Time-shared Bus**

### 2.4.4 Multiple Time Shared Bus

**A Multiple Time Shared Bus**, Fig 5, provides several alternative paths for a master device on the system to access a slave. It is understood that there is in the system some form of circuit that can inspect a bus, decide on its busy/free state and if necessary transfer onto another bus. The redundancy on such a system must be greatly increased by the increased number of alternate paths. In order to decrease complexity it is possible to assign particular devices to a bus, but this will detract from the redundancy offered by the system.

An example of this type is the Plessey System 250 Real Time computer, which Plessey say they designed with high system integrity and redundancy in mind. The controller of such a system must necessarily be complex and expensive, but it is possible to incorporate within this controller such features as priority marking of both busses and devices, and error recovery procedures.

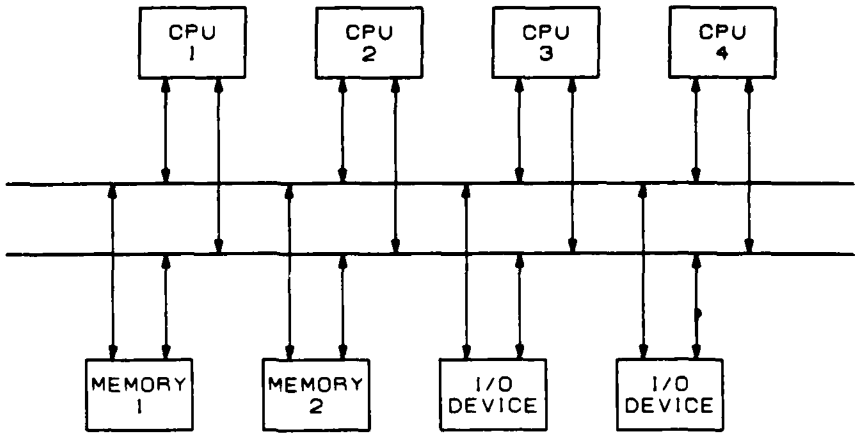

**FIG 5. Multiple Time-shared Bus**

### 2.4.5 Crossbar Connected System

If the Multiple Time Shared Bus is taken to extremes, it becomes the **Crossbar Connected System**, Fig 6. It is a method of connecting a number of CPUs to a number of memories and peripheral devices on a one to one basis. An analogy of this system might be a telephone exchange where subscribers are connected by a set of switches. These switches must necessarily be complicated when connecting elements of a multi-processor and there must be a cost penalty when implementing this type of approach. It is possible to have several pathways connected at the same time giving the crossbar system a high degree of parallelism and thus a high throughput. Another disadvantage besides the cost is that the system is very sensitive to crossbar switch failures, thus preventing 2 elements of the matrix being connected. To overcome this problem would mean duplication of, if not all, then those crossbar switches that would cause maximum disruption to the system.

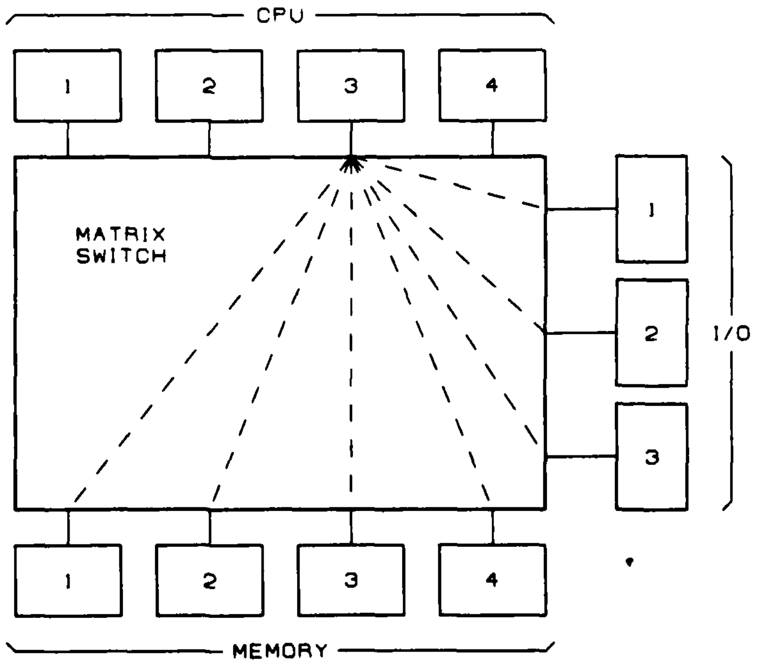

**FIG 6. Crossbar Connected System**

### 2.4.6 Star Connection

The **Star Connected System**, Fig 7, is generally applied to geographically distributed computers being connected via some form of data link to a centralised processor. The centralised processor is usually the control processor in the network using the resources within the other processors on the points of the star. As well as being the control processor, this processor may also be a data base controller providing a "library" function, with all the other processors acting autonomously. When the system is locally distributed, that is each module being within a few feet of each other, this central processor can be pure store, or intelligent store, providing a centralised data base and message passing device between the other processors.

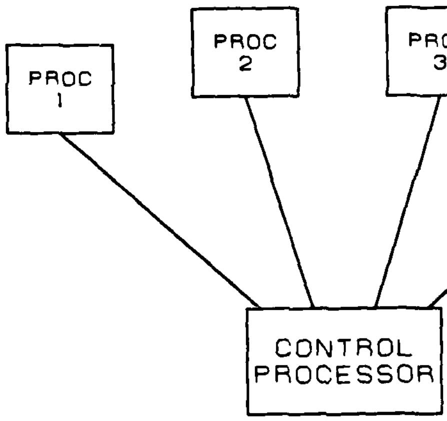

**FIG 7. Star Connected System**

In the locally distributed system, the coupling can be made very tight and then the difference between the star connected system and the dedicated bus becomes virtually nil. DISCUS is such a system, although DISCUS does not have a central CPU only a central store. Reliability must be a problem with the star connected system, which can be solved best by duplication - see the multiple bus above. The central store can be partitioned into several sections to provide this duplication but this leads to more complications and problems with function mapping at system build time due to allocation of these separated stores and providing a balanced allocation of functions.

### 2.4.7 Ring Connected System

Each processor in a **Ring Connected System**, Fig 8, is connected to a high speed transmission link that flows uni-directionally in a ring round the network. The problems that occur with a ring system must be primarily connected with the interfaces to the ring and the types of message passed round the ring.

This type of connection has been utilised in many places, and many examples of this type have been presented [6]. It offers considerable advantages in both reliability and ability of module extension. Reliability is enhanced by having 2 rings or so designing the ring interfaces that they become transparent at failure. It has been shown that this type of network is very suitable for micro or mini networks, because of its relatively low cost [7].

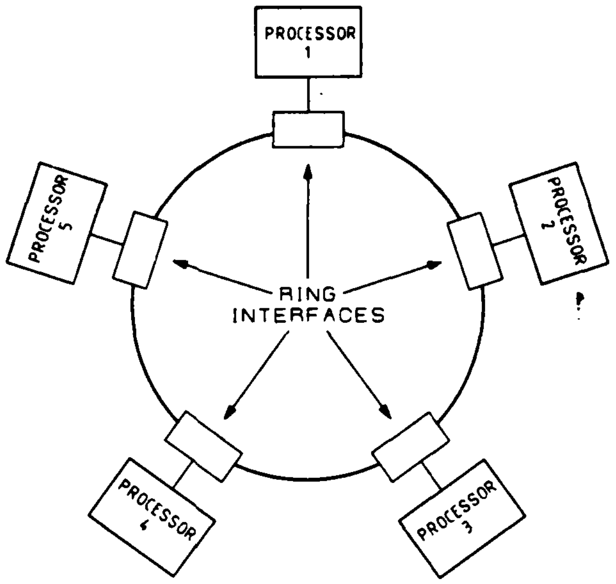

**FIG 8.** **Ring Connected System**

We are now in a position to look at the relative merits of all the above systems and try to derive an approach that is best for a DISCUS machine. Table 2.1 gives a summary of what I have discussed in detail above.

## 2.5 THE DISCUS ARCHITECTURE

Having discussed all the above architectures, we have to now consider what is best for DISCUS, or even if a radically new architecture is required. Initially the idea with DISCUS was to get some form of hardware working as quickly as possible. Simulating the system, while obviously useful, was ignored in order to get a feel of the hardware problems as early as possible since these would undoubtably effect the software.

Therefore a radically new architecture was out of the question. I maintained that any "special" components, such as bit-slice processors, should be rigorously avoided and normal commercially obtainable components should be used throughout. I was thus constrained to use a fairly conventional architecture for the individual processors. Let us review what we wanted from a DISCUS computer.

|  | DISADVANTAGES | ADVANTAGES |
|---|---|---|
| Dedicated Bus | Single point failure. Access from one processor to another. Bus contention high. | Cheap. H/W commercially available to grow system. |
| Multiport | Access to another's local memory. Very complex H/W. Many connections. Memory contention high. | H/W redundancy. |
| Multiport + local store | Complex bus interface. Many connections. | H/W redundancy. Contention eased. |
| Single shared bus | Access to another's local memory. Bus contention high. Single point failure. | Few connections between modules. I/O accessable by all. |
| Multiple time shared store | Access to another's local memory. Each card has complex interface. | No single point failure. I/O accessable by all. Few connections between modules. Dynamic reallocation possible. |
| Crossbar | Access to another's local memory. Very complex switching. Bus (cross bar) contention high. | I/O accessable by all. Dynamic reallocation possible. |
| Star connected with central controller | Single point failure. I/O not re-allocatable. Lots of cables I/O not accessable by all. | Simple interfaces. Isolation of processors. |
| Ring connected | Ring interface potentially complex. | Redundant connections. Suitable for local area networks that are geographically distributed. |

**TABLE 2.1 SUMMARY OF MULTIPROCESSOR TYPES**

The first thing is that we did not want to do any complex arithmetic. The power of an array of processors being used as an arithmetic processor is the ability to carry out a large number of parallel processes. In most cases this number is very high due to the types of calculation being performed: it could be as many as 256 or more. This gives an apparent increase in computing speed. It is particularly suitable for mathematics that involve large numbers of iterative calculations, for instance in weather forecasting. Accuracy of the basic quantities are the most important factors that these machines need. The speed of response to outside stimuli is not usually critical; there are no real time events outside that have to be recognised and acted upon in real time - although with weather forecasting it would be handy to have the results of the forecast before the weather actually arrived (but this is not what I mean by real-time here). The ICL Distributed Array Processor is an example of this type of computer.

We intended that DISCUS should be used a control computer for real-time applications. DISCUS needs none of the very fast computing speed or accuracy of numbers that complex real arithmetic imposes. Where control of external hardware is required, accuracy and speed are not always the overriding factors in the choice of the control element. Generally the ability to resist failure and to reconfigure itself to make these failures as transparent as possible to the user are the predominant factors. With DISCUS it was found that a microprocessor based processor would be quite adequate for the applications for which it was required. There is virtually no arithmetic processing in the DISCUS applications, in general only logical manipulation of data is required. Obviously any increase in processor speed would be desirable, but it is not essential to many tasks. Our initial overriding interest lay with the operating system and its recovery mechanisms. In the discussions that follow the type of tasks that DISCUS is designed for must be borne in mind before any judgement is given. These tasks require the control element to be reliable and resilient to failure: an example is circuit switching in a communications network.

When we looked again at the aims of DISCUS (see section 1) we decided that the third item had been sufficiently well explored through current (1977) research. What had not been investigated was software recovery and reliability and the help the hardware could give to the operating system in achieving these aims. Although several other benefits and interests were found on the way, initially there were only two major points that interested us:

i. Improve the containment of errors.
ii. Reduce the software complexity if possible.

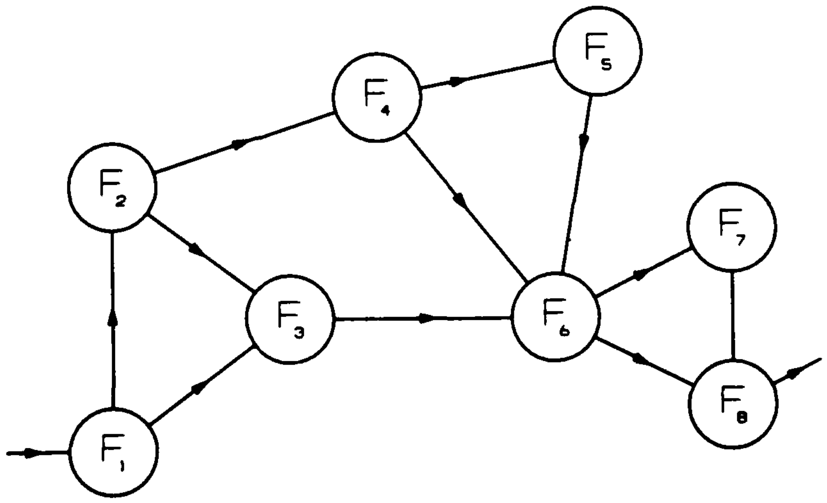

**FIG 9. The Functional Approach**

As I indicated at the beginning of this section we adopted a functional approach for DISCUS. What was required was a set of functions, Fig 9, mapped onto a set of individual (local) processors each communicating together via predefined protocols in order to perform a single job. An immediate advantage advantage of separating functions onto their own processors is isolation. This helps to contain errors to a single local processor.

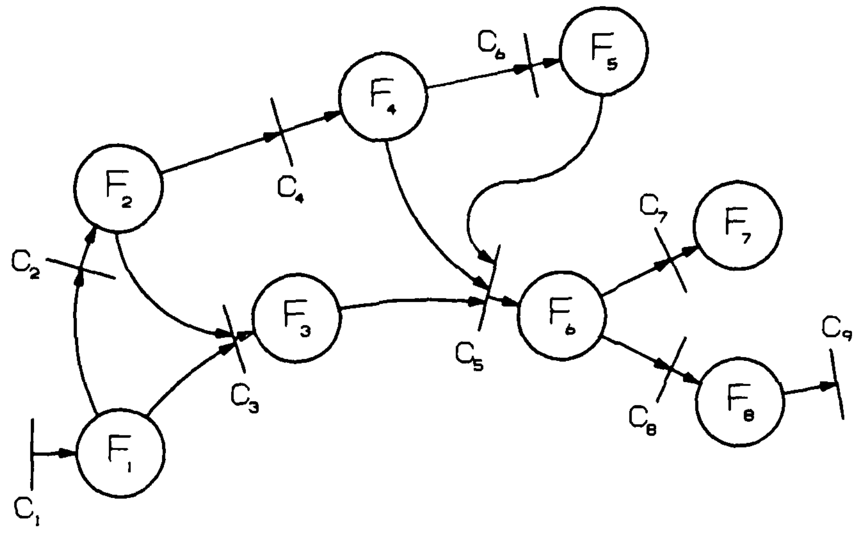

**FIG 10. DISCUS Message Passing**

Each function must be able to communicate with its designated fellows. There are two sorts of communication media that are required; message passing for the basic transaction, and shared common store for the processing of that transaction on the functions.

Ideally, in order to contain errors, messages should be passed via physical channels connected electrically only to those processors concerned, and with physically restricted access (i.e. Read Only for the destination processor), Fig 10. Shared store should be accessed via similar restrictions, Fig 11. However it is not as simple as it may seem to provide this type of architecture.

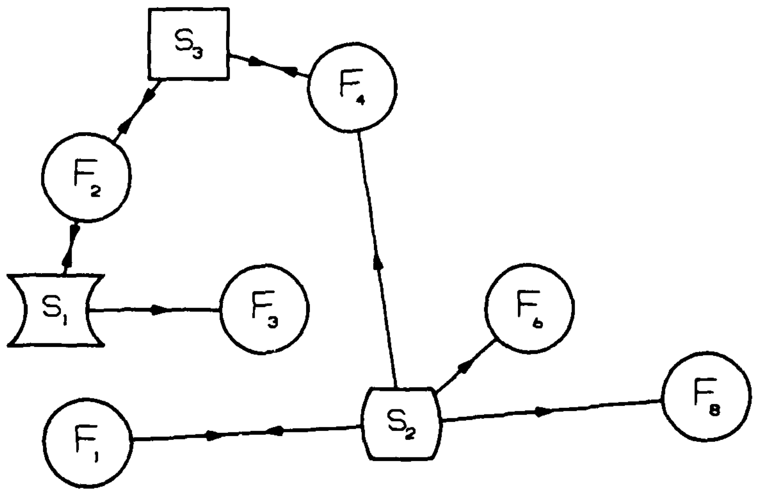

**FIG 11. DISCUS Shared Store**

## 2.5.1 Separate/Common Channels

By having separate channels between functions which connect only those functions that need to communicate, there is no possibility for a function passing messages to the wrong place. These would consist of individual cables that connect only those functions that the original functional split dictates. However there are many disadvantages to this method.

The first is that we lose any ability to change the system quickly. In a research environment this is highly desirable. It is likely that there will be several systems to be run on the same machine. Unplugging and re-plugging the system will be tedious, prone to mistakes and introduce a further point of failure in the system. If there are a large number of processors, there will a very large number of connections to be catered for.

If n is the total number of processors:

then $\sum (n-1)$ is the total number of connections to be provided.

and n-1 is the total number of channel ports on each processor.

For instance in a 6 processor system there will be 15 possible channels. Each processor would have 5 connection points. For a 32 processor system the number of connections would be 496 and 31 ports. The number of connections is obviously absurd. It could be argued that only those connections that are needed are provided. In a mature stable system this can indeed be done. However with the experience of the present application of DISCUS (a telephone exchange) even this might prove difficult. If some form of central error reporting or supervisor function were to be used then this one would require as many ports as there were processors, so the above numbers do not appear to be very contrived. When the system was being used a research tool it is extremely undesirable to have separate channels. Each time the system was reconfigured the channel cables would have to be rewired. Thus there would still be a large number of connections and the rear of the equipment would become festooned with cable.

The channels would have to be serial rather than parallel, since the amount of wiring and number of individual connections would be prohibitive. The reliability of all these connections would be very low. If the serial scheme is used, a communications protocol would have to be developed (or borrowed, like IEEE 488). If a standard serial bus was not available (and even if it was) circuitry would be needed to interface to the each DISCUS processor. If a reliable error resistant channel were required a serial bus of the complexity of the new P896 serial bus [27] would be needed. From this experience of the new IEEE P896 serial bus this could be up to 50 MSI circuits. Even if the full number of channels was not used each processor would require all the ports. With a 24 processor system the number of circuits would be 23 x 50 - a substantial number. Even if only a quarter of the ports were used the amount of circuitry is prohibitive. It would be possible to produce a custom LSI circuit of the controller but this would take time, and be a special item, and go against the spirit of DISCUS as discussed above. A simple serial scheme like the V24 interfaces to VDU's etc could be used, but even that would require several devices to implement.

Perhaps the biggest disadvantage is when it is required to implement one of the strengths of DISCUS. Duplicating functions I have shown to be one of the most important features of DISCUS. With separate channels it would be extremely difficult and take complicated hardware to provide t is facility.

Ideally the likely *functions* to be duplicated must be identified before the system is built and provision made for this with the hardware. To retrospectively add another duplicate function would be virtually impossible.

There is one advantage: the inter-function communication is faster. Despite all this however, we felt that separate channels was something that should be ignored.

### 2.5.2 Separate/Common Global Store

Arrays of data that are shared between two or more functions should also ideally be connected to only those functions that require to access the arrays. However the problems of having separate global store are the same as the channels, only more so. In order to achieve a reasonable speed for transferring large data arrays these connections should not be serial. Now that these connections are in parallel the problems of the individual interfaces become even worse than the serial connections. Thus I do not intend to dwell on this subject further.

### 2.5.3 Store Partitioning

If it is decided that there is only one common store with a common bus as the present DISCUS, there are still one or two tricks that we can try to go part way towards separate channels. It would be possible to split the common bus into sections with one or two processors on each section, together with a portion of the global store. In between each section of the common bus would be some form of "gateway". This would allow a processor on one section of the common bus to access a global store on another. The design of this gateway would be complex and how it would be used and linked to the system generator would also be difficult. Because of this it worth asking whether it is necessary to split the store bus in this way.

The reasons for splitting the bus are various:

i. To separate channels and common data.

   By having pairs of processors (or single processors) on each common store bus. If a processor wishes to access another processors global store, it might have to cross several gateways to reach it. Mapping a system onto this "paired" architecture might not be very convenient and the functional split might be constrained by the architecture - something that is not wholly desirable.

ii. To duplicate the paths to store to relieve global store contention.

   Only if the functional split has been chosen so that each of the processors spends more time accessing global store than local store will the extra global route be helpful in relieving global store contention. Applications where very large arrays in global store are continually accessed might be another area where this might help. At present with the current applications tasks, the main constraint on speed seems to be the amount of work that the operating system does. I have found that the overhead in accessing common store is negligible. It does not afford any reason to duplicate the common bus.

iii. To duplicate the paths for redundancy.

Possibly the strongest reason for duplicating the common bus is for redundancy. The chance for single point failure within DISCUS due to the common store is high. By having a duplicate path this risk would be reduced. However as I said redundancy is fairly low in the list of desirable features at present.

The problems of having many physically separate stores and channels, which in a research environment would need to be continually changed whenever the software was changed, I considered to be a further complication we could do without, and so we decided to start with a simple method.

Of all the above methods of connecting processors together the simplest is the star connection. Its advantages are -

i. There is a centralised arbiter, which makes it possible to control the hardware polling to allow fair access to the central global store. You must remember that all the processors are equal, there is no control processor. It would be undesirable to have any sort of priority system, where one processor could get locked out.

ii. The interface to global store is relatively simple.

iii. The global store becomes a tightly coupled extension of the local store, so that producing code to access global store is simple.

iv. It is impossible for one processor to corrupt another's local workspace store.

There are disadvantages however -

i. It is possible to have single point failure. This was not considered to be a good reason for not using this method. Redundancy of hardware, although an original aim, was soon abandonded for the first hardware in preference to the other aims.

ii. The input/output to external devices are not accessable from all processors. It was intended only to have a passive store in global store with no processors or I/O connections there. The I/O would be connected to a single local processor.

iii. There are potentially lots of cables connecting each local processor to global store. I will come back to this problem when discussing the hardware.

What we have ended up with was not the star connected system I defined earlier. We have now lost the centralised control processor, and now only have a passive message passing/common data store. This store would have its access restricted via a combination of hardware and software.

At present <u>all</u> the channels reside in a common global store. This store is the same as the common data store where all the object arrays are placed, (Fig 12). There is no inherent protection on these channels other than that provided by the operating system.

It has been the aim of DISCUS to make the hardware and the software as simple as possible. While in some places this has only been barely achieved, the bulk of the system is felt to have met this original intention. It is essential that any new machine should keep to these aims. All the components should be "off-the-shelf" items. There should not be any need to use custom made components. Also technologies such as bit-slice processors should be scrupulously avoided: they are far too complicated and can cause too many opportunities for errors at the basic level. The "bit-slice" processor approach affords a marvellous opportunity to ruin the whole system if insufficient care is given to the design, and diverges from the DISCUS principle of keeping things simple.

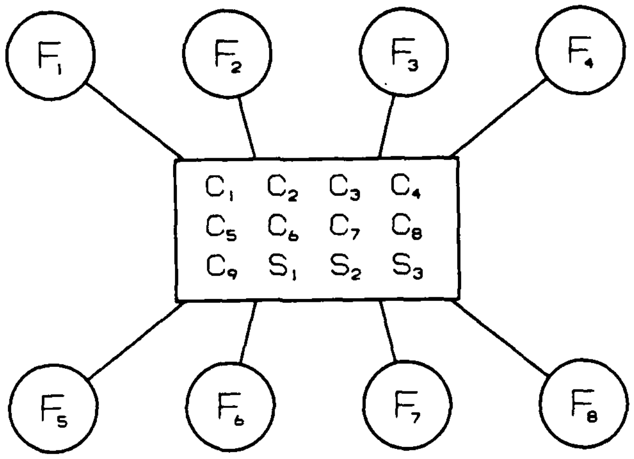

**FIG 12. <u>Physical Connections</u>**

In most practical applications the operating system and the hardware can protect against most accidental faults. It is impossible to protect against the determined deliberate saboteur. It only requires an endless null loop placed in a program to stop the function working, and although there are software validation methods, these do not ensure totally fault free software. By having the hardware physically to restrain the programmer, even if he does make a mistake, the results of this fault are contained locally. By having a small, secure software kernel with the hardware the applications programs can be run with a large degree of protection.

Within DISCUS there are several mechanisms at present that help with the security of the system. At present each function of the system is partitioned onto a separate processor. Each function is unable to interfere with another function, either deliberately or accidentally. On any new machine this would be an important concept to retain. It offers both the ability of secure functions, and the opportunity for duplication and redundancy.

To sum up, the present configuration of having the processors on a global bus able to access all the global store equally is probably still the best scheme. The concept of having separate (serial) channels or partitioned store is undoubtably a powerful one, but presents difficulties in implementation that I do not consider worth the effort involved. It would go against keeping DISCUS simple and thus easy to verify.

There are still other ways that global store/channels can be protected. Originally we thought that DISCUS could use an "intelligent" common store. This was supposed to manage the global store and allow the local processors to address data objects by name. This would mean that virtual addresses would be carried on the global store, which would no longer become a simple extension of the local storage space. Care would have to be taken to ensure that a mixture of absolute and virtual addresses does not take place. To mix them would lead to much confusion. Bearing in mind the plea for simplicity, the difficulties that this presented ruled it out for DISCUS.

It was considered that some form of memory protection mechanism similar to the Plessey System 250 "capability" structure could be used. This would allow the operating system to protect global data (and local if necessary) by giving access rights to the various global objects. The applications program would have to go through this mechanism in order to reach any global information or send or receive messages down the channels. Array overflows and read only arrays can be catered for, as well as illegal accessing other channels. By keeping the applications programmer away from the global objects directly a large measure of integrity can be given to these objects. To implement such a scheme would take a lot of logic, and since there were no LSI memory management units available, it was decided that this was one thing that would be held over to a future machine. This all meant that there was a considerable burden on the software to check all the array bounds and read only types on the arrays in software. It is thus this code which slows the system down when accessing global store.

## 3. THE DISCUS HARDWARE

As I explained in Section 2, DISCUS is essentially a star connected system, Fig 12. Each processor in the DISCUS array can access a common global store as well as its own local store. The global store can be up to 7 blocks of 32 kbytes of store, and up to 64 kbytes of store for each local processor. The program code is stored in local store so that normal processor working is carried out in parallel with the other local processors in the system. It is only when a local processor wishes to access the global store, for either common data or to pass messages to another processor, that any contention between them might occur. These requests to global store, and all DMA requests on each local store are carried out completely asynchronously. This relieves the physical restraints on the interconnection of the various modules within the system since there is no common synchronising clock, with all the problems of clock skew that synchronisation brings.

The hardware is designed around a bus system that was developed especially for the DISCUS multiprocessor rather than use an already existing one. At the time of the original system design there was only the S-100 and the INTEL Multibus busses available for use in microprocessor systems. The S-100 standard is an American based one, developed for, and by, the amateur computing market. This standard is not as well defined as it should be, not having a central organising body to monitor the standard, and it is totally unsuitable for DISCUS, although at the time of writing the IEEE has produced a proposed standard for the S100 bus that is awaiting general agreement. The Multibus is a commercial bus produced by INTEL in America for use on their MDS-800 range of microprocessor development systems, and, although it is relatively well defined compared to many, at the time it just did not do what was wanted for DISCUS. DISCUS requires some specialised control signals such as "Memory Locked" and "Write Protect" in order to make it function properly. The bus that was finally developed for DISCUS is by no means perfect, but it has served to get DISCUS into use; and it is also worth noting that it has not been thought necessary to add to the bus throughput the development of DISCUS. Most 8 bit microprocessors can be used on the system (at present the Z80 and the 8080 micros have been running on the system), but unfortunately the newer 16 bit microprocessors, such as the 68000, Z8000 and the 8086 will have to have new memory cards etc to allow correct operation.

In all honestly I must say that the present bus, in the light of what we have tried to do with it, is a shambles. Not a great deal of thought was put into it and we are now paying the penalty. For this reason it would be necessary to choose a very well defined bus for the next DISCUS, if possible a standard one. I have looked at the problems and attributes of busses in a little more detail in a following section and I will leave the subject for the moment. A brief description of the present DISCUS bus is given in the Appendices [29].

### 3.1 LOCAL PROCESSOR DESIGN

By modern standards the local processor design is fairly conventional. Each section - ROM, RAM etc, is separated on the bus physically, rather than put on the same card as the processor. It was felt that this allowed a greater flexibility of how the processors were arranged, although another consideration was the packing density of the double Euro-cards that were used. With these cards it would have been impossible to fit the required logic for the current implementation of DISCUS into a single printed circuit card.

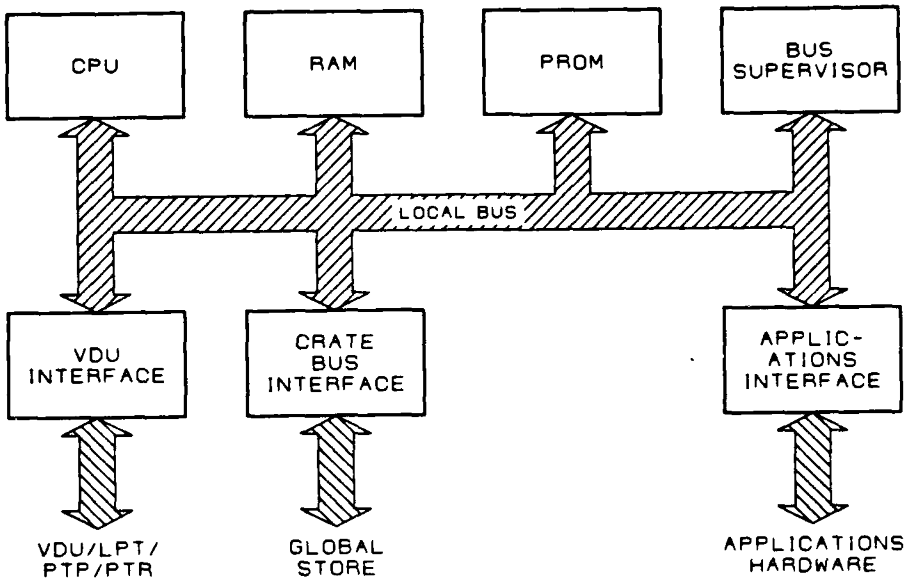

**FIG 13. Local DISCUS Processor**

Fig 13 shows a block diagram of the local processor. None of the features offered are particularly novel, except perhaps for the asynchronous controllers. On all possible occasions the maximum use is made of the specialised LSI circuits offered by the microprocessor manufacturers. Up to 64 k of store can be used on the local bus. This can be a mixture of RAM and PROM over the entire range. There is also a small amount of PROM on the processor card called the BOOTSTRAP PROM. This area of store is not part of the main store but it is a "shadow" PROM. This means that it is there for only as long as it is needed during the setting up of the system and then it becomes unavailable, being replaced by the main local store after initialisation. By having this piece of store it is possible to set up the various interfaces on the processor card and other peripheral cards without having to alter the applications program.

There is a Bus Supervisor Card that provides a check of all the local bus activity and can detect simple bus faults. From the start, DISCUS has been concerned with fault detection and the recovery from those faults. With these points in mind, fault detection and the alerting of the applications programs and the operating system goes right down to a hardware level. There is a heirarchy of fault detection within DISCUS, and it is important that the correct recovery action is taken at every level. For instance, at the hardware level the detection of faults should only alert the operating system and not the applications program. This allows the applications programmer only to deal with applications induced failures. He must put in his own recovery mechanisms for these failures, after all, he probably knows best. This leaves the operating system errors to be coped with at that level where possible.

The DISCUS bus is a fully handshaked system; every command that occurs on the bus must be acknowledged for correct operation. Failure to do so can be from several causes:

i. The unit being addressed (store etc) could have faulted, and is unable to reply.

ii. The unit being addressed may not exist.

iii. A write command could be presented to a read only store, or a write protected area of store.

All these faults will appear as a failure of the bus acknowledge system. The Bus Supervisor Card is able to detect these; indeed at present this is the only fault it can detect. It can provide some form of action to enable recovery to be carried out. It is also possible to access the bus via this card in order to monitor activity on the bus using some form of passive Logic Analyser.

Much use is made of cheap "intelligence" within the DISCUS system. The peripheral controllers are provided in a variety of forms. The VDU is controlled by another microprocessor, along with a paper tape punch and a reader, and a serial line printer on the same card. (See the Appendices [29] for a description of the program and peripherals currently in use for the peripheral card.) Using this method many of the peripheral dependent actions can be done on this card rather than on the main local processor. It contains its own memory and I/O ports and accesses main store by an asynchronous DMA scheme. On the present DISCUS system, some of the facilities provided by this card are: line editing for the VDU, and punch information such as "tape low". Because the inputs from the VDU normally are assembled into lines that are passed via a predefined software protocol, it is possible to change the peripheral card to cater for different types of peripheral without having to alter the local applications programs or the operating system. Also the local processor is only alerted when a complete, checked input line is ready.

All the above cards can be used as a single processor facility, plugging into a standard prewired backplane with a minimum of restrictions on where cards may be placed on that backplane. The system is based on the standard double Eurocard board, so that standard commercially available racking can be used throughout.

## 3.2 MULTI-PROCESSOR ARRANGEMENT

Each local processor is able to communicate with a global store in order to store common data and/or to pass messages to other local processors. It was felt that when the number of local processors exceeded about 6, the number of cables interconnecting the system would become excessive, since each local processor requires two 50 way ribbon cables for connection to the global store.

In order to reduce this quantity of cables when large numbers of local processors are used, a scheme of using a "concentrator" in each local processor crate was adopted. Fig 14 shows the arrangement of a single DISCUS processor crate with 3 local processors. There is a subsidiary bus, known as the CRATE BUS, that joins the 3 local processors together. The allocation of this bus is controlled by the CRATE BUS CONTROLLER. This card provides asynchronous arbitration of requests from the individual processors for global store. This method allows a maximum of 4 times the number of local processors as there are interconnecting cables. In the current DISCUS there is a maximum number of 3 local processors in each local processor crate.

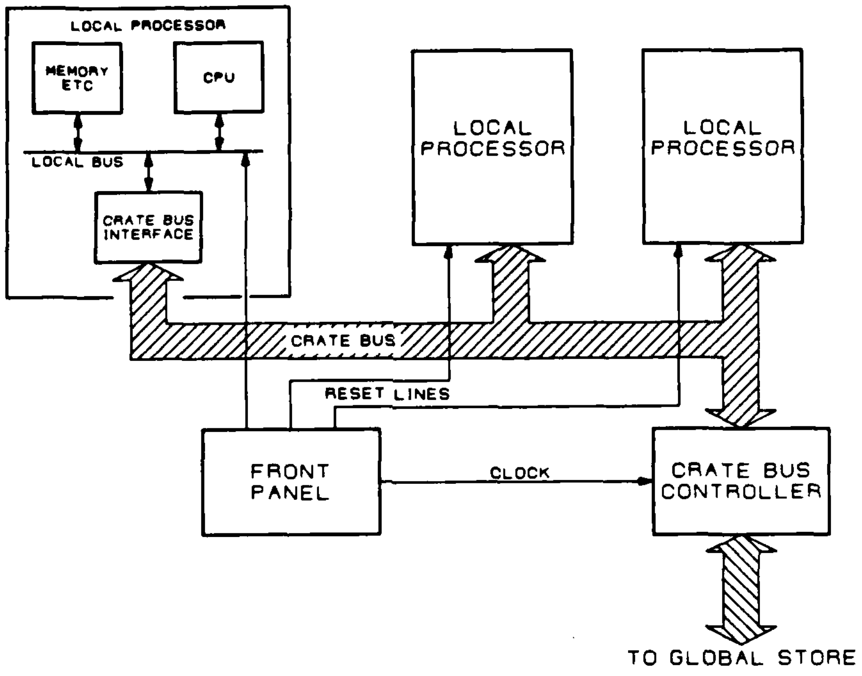

**FIG 14. DISCUS Local Processor Crate**

The crate bus is formed by a pair of 50 way ribbon cables that run along the front edge of the cards. Plate I shows a local crate of 3 processors with its global store crate; the 2 ribbon cables for the crate bus can be seen across the front of the local processors in the top crate. Although this would seem to prevent easy access for inspection of individual processor cards on extender boards, this restriction is not as bad as it might seem. Plate II shows a complete DISCUS multiprocessor with 8 local processors. One of these processors is an MDS-800 development system that acts as the system loader and also the disk device handler.

Since DISCUS is concerned with distributed hardware as well as software, this implies well defined interfaces between processors, and to the outside equipment being controlled. This allows the specially designed external interfaces to be fully developed on a separate single DISCUS processor with no front crate bus cabling. A minimum of testing should be required when this is transferred to one of the local processors in a multiprocessor DISCUS. If it is required to extend cards with the crate bus in place, it is possible to fit an extra long crate bus cable so that the extender cards can fit.

Each processor in a local crate is user controlled from a common front panel at the right hand side of their crate. This has the reset buttons and the fault lights for each local processor within that crate. There are also indicators for each power rail within the crate. Connection to the global crate is taken via a pair of ribbon connectors at the rear of the local crate.

PLATE I. Local Crate with Three Processors

**PLATE II. A Complete DISCUS Computer**

Note: This shows a 8 processor system with an Intel MDS-800 minicomputer as a ninth processor for system loading and as a disk handler. The blank space is for application's hardware.

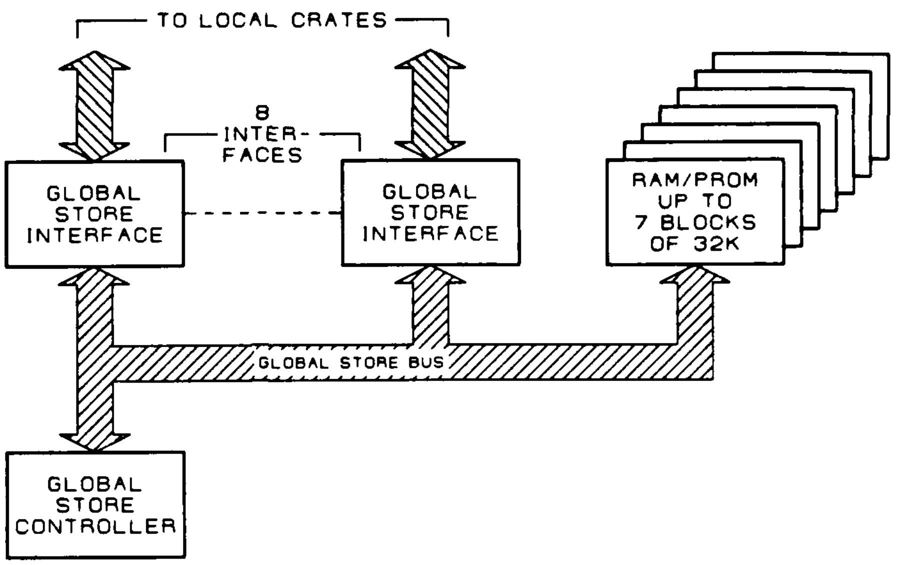

**FIG 15. DISCUS Global Store Crate**

## 3.3 GLOBAL STORE CRATE

The global store consists of up to seven 32 kbytes blocks of store. This store can be either RAM or PROM fully interleaved down to blocks of 2 kbytes. The global store bus is identical to that on the local processors, except for the DMA request lines, so that all the memory cards and supervisor cards can be used on this bus with complete freedom. However, there is no facility for input/output on this bus, as it was felt that the I/O should be handled as a distributed function rather than a global one.

Each local crate (up to 8) is connected to the global bus via a GLOBAL STORE INTERFACE, Fig 15. This interface contains all the necessary cable termination logic for the ribbon cables from each of the local crate bus controllers. This is a global store controller that arbitrates between the asynchronous requests from each local crate. The main global resets common to all the processors are generated on this card, which also has the power indicators for the global store crate.

## 3.4 DISCUS ADDRESSING SCHEME

In order to address its own 64 k of store and 7x32 k of global store, each local processor has an extra 3 bits on top of its normal 16 address bits that most common 8 bit micros use (8080, Z80, 6800, 2650 etc). These 3 bits are set in an Auxiliary Address Register (AAR) as a page number of global store, where each page is 32 kbytes. These page bits are set by addressing an output port on the local processor card. So that this port is not being set continuously, the most significant bit of the normal 16 bit address is used to envoke these extra page bits. When the msb is logic zero, the page bits have no significance to the complete address and normal addressing can be carried out over the first 32 kbytes of local store. The 3 page bits can be set to anything since they are not used. When the normal msb is logic one, the 3 page bits are used to make up a complete 19 bit address. When these bits are all zero, the processor addresses the top 32 kbytes of local store; while any other value of the page bits addresses the appropriate page in global store.

The convention that as been adopted for the page number is as follows:

| ADR15 | ADR16-ADR18 | RANGE | NAME |
|---|---|---|---|
| 0 | X X X | 0 k - 32 k | Local 0 |
| 1 | 0 0 0 | 32 k - 64 k | Local 1 or Global 0 |
| 1 | X X X | 64 k upwards | Global 1 - Global 7 |

Note: i. Address bits start at bit 0.

ii. Local 1 and Global 0 are alternative names for the same address range.

It was felt that this scheme was good enough for the DISCUS system despite its disadvantages. It allows for large amounts of data storage in global store, which is a prime requirement of the systems for which DISCUS might be used.

Two other signals are produced by the AAR - "Memory Locked" and "Write Protect". Memory Locked is essential to produce the indivisible operations in global store so that semaphoring on global objects can be done correctly. Write Protect is used by the operating system for read-only arrays in global store.

One disadvantage of this method is that the global store is no longer linear. Since bit 15 is used as a page select, even 32 kbyte blocks of global store are unavailable. For instance:

28000H - 2FFFFH will do operations in global store page #2
40000H - 47FFFH will access 0000H to 7FFFH in the local store

The fourth digit of the 5 hex digit address must be greater than or equal to 8 to select global store, since this ensures that bit 15 of the address is a logic one.

This is however one clear advantage that this method has that makes it very attractive to us. It is impossible for any code that is in the top half of local store to access directly the global store. In order to access global store the AAR paging bits have to set to non-zero. This action then removes the top of local store, and the code that was running effectively vanishes, the results of the operation being somewhat unpredictable. The operating system makes use of this overlay by putting the applications programs in this space, thus making the applications programs go through the operating system in the lower half of store to get at the various global objects. There would still be ways of accessing global store if the applications programmer was to write a program to the lower half of local and then run it. This eventuality has been sorted out in the DISCUS Mk 1.5.

## 3.5 SYSTEM LOADING

In order to develop programs for the system, a set of compilers and linkers are used on a separate host computer. There are 2 methods currently being used to transfer the assembled machine-code programs onto the DISCUS multiprocessor.

i. By programming a set of PROMS and using a PROM card for the main code storage in local memory. This is a very time consuming process, and if it is required to make a change, however small, the whole process of erasing and reprogramming has to be carried out: which procedure can take several hours. Only when a final system has been successfully run over a period of time and confidence has been gained on the integrity of the system and all its various programs, should the software be committed to PROM.

ii. By loading via some external means into RAM. This allows the programs to be loaded much quicker.

At present, DISCUS can use 2 methods of type ii program entry. The first is by paper tape produced by the host computer and entered using a reader connected onto the VDU peripheral cards. A MONITOR program, see the Appendices [29], was developed for DISCUS that allows a limited amount of user control over the system via a VDU. One of the facilities provided permits the reading of a paper tape formatted to INTEL's Hex format [25] into memory. This has proved particularly useful in debugging the hardware and commissioning new cards into the system, and also for running small programs on either a single processor or one or 2 processors in a multiprocessor. Also it has been used for getting the initial applications and operating system software running.

This scheme obviously requires that there is a VDU card, VDU and paper tape reader on every processor within the multiprocessor which just is not practically or physically possible. A better solution is to put a simple bootstrapping routine into PROM on each processor and load the programs via global store using only one processor to do the loading from the host computer. Rather than use paper tape, it was decided to load the programs from the same backing store medium of the host - floppy disk. In order to load this, it meant that a floppy disk unit had to be interfaced onto one of the DISCUS processors. This requires that a special card be made to interface the disk drives onto the DISCUS bus, along with a considerable amount of software to control the transfer of data to and from the disks. A more satisfactory solution was found to be in turn the host processor (an INTEL MDS-800 minicomputer) into a DISCUS processor. This enables programs to be loaded directly onto DISCUS after they have been developed, thus saving a considerable amount of time. Only one card was necessary in the MDS-800 to provide the correct interface between the INTEL Multibus and the Global Store crate. The system that was used is described in the Appendices [29].

## 4. RECOVERY IN DISCUS

In any piece of electronics hardware there will inevitably come a time when it will fail. This time may be of the order of years or may be as short a time as hours when the equipment is in a harsh environment. The failures that occur may be in a single component or may affect a whole group of components. They may be catastrophic, when the failed components will be obvious from a visual inspection (or smell), or the failures may leave the equipment physically intact, but electrically wrong. All these faults must be detected and cured. In almost every application that does not include a computer or does not rely on the equipment to give uninterrupted service, the diagnosis and repair of the equipment is usually a manual process. The full testing of the item can be done either automatically or manually, the former being used when there is a large volume of testing to be carried out.

### 4.1 FAULT DEFINITION

Before going any further it is essential that some form of definition of a fault is made. Until we know any better, the following short definition is made:

> A fault is any deviation from the required operation of the equipment.

In saying this there are immediately a number of difficulties inherent in applying this definition to a computer system. The primary difficulty is concerned with the phrase "required operation", because this implies that the required operation has been completely and correctly defined in the first place. There are several points in the design sequence where it is possible that faults will be generated that prevent the final equipment from behaving in a manner that the original designer did not want. There must be many occasions where customers try to rectify what they consider to be faults on their equipment which are in fact quite valid design decisions by the manufacturer because the customer did not define properly what he wished the equipment to do. It is essential therefore that when an equipment is designed, the correct definition is made of what is required from the final equipment. It is only by knowing exactly what it should do that it is possible to see when it deviates from its design specification.

Thus our original definition of a fault can be altered as follows:

> A fault is any deviation from the required operation of the equipment as defined by the original requirement.

We can split this definition further:

> **A Design Fault** is one that is caused by an error in the design process. It can be from incompetent knowledge of the job being done, or from an inability to understand the original design requirement.

> **A Transient Fault** is one in which no immediate replacement item of software or hardware is required, although it will require some automatic recovery action.

**A Permanent Fault** is one in which some form of replacement item is required to achieve the correct overall operation.

There are both transient and permanent faults through all the various stages of design, manufacture and use. The hardware faults can be either design faults or ones that occur due to failure of some component. The failure of this component is an unforseen event for the design engineer and he can do nothing about it (except to ensure that his equipment is generally robust enough to cope with failures - no mean achievement). ALL the software faults will be design faults - there is no such thing as component failure in software, thus the system must be robust enough to cope with these faults. This is done by forcing disciplines on the programmer by the way he writes the program. The operating system can provide standard protocols that force a way of writing, and the programmer can adopt defensive programming to check for errors. DISCUS was initially intended to be a hardware redundant machine as well as having a standard operating system. However the former idea was abandoned in favour of the software because it was felt that hardware recovery/redundancy techniques were relatively well explored, while software recovery was not.

## 4.2 THE DESIGN PROCESS

It is the task of the design engineer to take the specification that has been produced and turn it into a design for real equipment that can be handled by a team of engineers whose job it is to produce the real equipment. We have a hierarchy of tasks that have to be controlled in a very defined manner in order that the correct implementation of the original intention is produced. There are a considerable number of potential places at this level where faults can be introduced into the system. The majority of these are almost certainly bad design, with the rest being misinterpretations of the original requirement.

Any computer system whatever the size, must be specified rigorously from the start if a properly designed and maintained system is the goal. The functional specification of the system must be the starting point of any design. Initially this specification must be a total system description and of a relatively general nature. As the specification design proceeds, the specification is gradually broken up into a set of inter-related functions. This process continues until the point is reached where it is impossible or inconvenient to break down the design any further. These functional blocks are either simple primitive software operations, such as procedures, or the basic hardware building blocks discussed earlier. A large part of the design is concerned with the various interactions between all the software and hardware blocks. In designing a functional block it is desirable that the interactions with its neighbours are as simple and as few as possible. With real time computer control systems, a large part of the design process must be concerned with the problem of ensuring that when the various function modules interact, they only do so via the specified interface and not by accidental or deliberately wrong channels. Not doing so can result in system failure, especially in cases when a common data base is involved. This failure could be catastrophic, affecting part or all of the system, or it could subtly alter the systems response to stimuli. This latter is more difficult to detect and can be ultimately no less damaging than the catastrophic failure.

It is important therefore that the overall system designers make the standard interfaces between each functional block as rigorously controlled as possible. With interfaces between modules of software sharing the same store, this is obviously more difficult than with hardware, which has physically discrete connections. In order to ensure that the relevant software protections exist on a single processor/store computer, there has to be a complex operating system/scheduler, which must trap errors within the software; and obviously this operating system is going to be prone to errors like all software.

When someone designs a circuit or a program, it is normal that they would like to hold the entire thing in their head. By doing this a much greater appreciation of the whole design is gained and possible fault conditions can be isolated at a very early stage in the design process. When several people are involved in the design of a single highly interconnected piece of equipment that uses both hardware and software it is impossible to predict the behaviour of it when only part of the final equipment is under one person's design authority. As a result, when the final testing is done, there can be a multitude of interface problems that can take a great deal of effort to put right. The only way to really overcome this is to have a rigorous approach to the design specification. The interfaces must be as simple, well-defined and obligatory as is possible. Divergence from these must be allowed only as a total last resort. DISCUS allows this to happen quite naturally with its concept of one function per processor, although the channels and common store between the functions must be no less well-defined and adhered to. As I have said before the best way of thinking of the power of a multiprocessor like DISCUS is that it provides each person with the opportunity to have control over a "brain-sized" package of software that he should be able to grasp in its entirety with simple interfaces to other functions (assuming however that the initial function split was carried out in a well controlled manner).

## 4.3 FAULT DETECTION

This phase can be the most difficult, since it has to encompass a wide variety of faults. These may be of the "its obvious because I have disintegraed" type to the "I think I will fault every 10 days for a few milliseconds". Both extremes present unique problems which must be catered for when including some form of mechanism for fault detection. In a large computer installation, it is hoped that the more catastrophic types of failure do not often occur. The most common are the single "quiet" failure in a single module, are nowadays probably a failure of the logic (caused by an integrated circuit).

The impact of this will be dependent largely on where the offending component is located. For instance if it is a store location it might be possible to continue using another part of store; but if it is the main bus circuitry, unless there is a degree of redundancy in the design, the failure will cause a complete failure of the system.

The faults can be detected at many levels of the software and hardware hierarchy; indeed it is essential that each level is equipped with its own detection mechanism if a successful attempt is to be made at isolating the fault.

At the highest level we have the applications programs and any specialised interface equipment that is being controlled by those programs. When considering the design of his software, the applications programmer must be aware of all the conditions that lead to failure at his level of the program. These faults must be as tightly contained as possible, so it is essential that the mechanism for detecting them is included in the design from the earliest from the earliest possible moment. The designer must try to prevent these faults from descending to the next level down, which is the operating system. It is possible to use any error detection features that the operating system offers, but the latter should not have to inform (or usually have the power to) the applications engineer of any faults in his software other than where it has to deal with system variables and some system hardware. At this highest level, there can be either hardware or software failures. The hardware failures resulting from the applications hardware can usually be detected by a combination of software and specialised hardware. For the moment, we will ignore these failures as they are chiefly the responsibility of the applications engineer, although it must be stressed that they must form part of the complete fault detection scheme. The software failures can be more transitory and elusive and it is thus essential that a well structured and disciplined approach be taken with the applications software (and at all levels of software). The situation is helped at present by the gradual introduction of languages that force this discipline on the user. One of the most common is PASCAL. With its highly structured approach, and the many methods of bound and variable type checking, a user can use this to produce a very tightly controlled environment for the running of his applications programs, albeit at the expense of more code and thus more time compared to the unstructured languages, such as FORTRAN. Although it should be noted that there exist hardware mechanisms for doing such bound checking that will alleviate most of these speed limitations (see below).

There seems to be at present an eternal quest for speed as if that was the solution to everyone's computing problems. While it must be agreed that speed (or the lack of it) is a serious concern, the ability to protect against deliberate or accidental failures and thereby the time it takes to recover from these failures must be considered to be a more dominant factor in the choice of machine and language.

It is within the operating system that the greatest burden of detecting and correcting faults is carried out. The operating system is able to detect a fault at its own level and to a limited extent within the hardware, although this will usually be via some indirect means because the hardware has caused a failure of some function or task and the protection mechanisms within the operating system have been alerted. Generally the operating system is unable to detect true applications faults except where these cause failures in data or message handling procedures. In most operating systems there should be extensive fault detection mechanisms that guard against invalid or unsuccessful operations, while being invisible to the applications programmer. At present with single processor machines, the bulk of the effort of both detection and action of faults must necessarily be done in the operating system. There is a hierarchy of levels within the operating system that each have the ability to detect a range of faults and report or cure these faults. With the advent of cheap processors, it has become possible to spread the computing power both vertically and horizontally in the hierarchy so that it is feasible to detect faults more readily and to place a lot more of the detection in hardware where before it had been in the operating system.

With the hardware, unlike the software, the possible faults tend to be both more permanent and more predictable: when a component fails it will usually do so more permanently, but software bugs can manifest themselves far more fleetingly only when a special set of circumstances arise. It would seem sensible therefore, that as much fault detection as possible is carried out at the hardware level where it is easier to isolate faulty areas. However, the hardware must be designed from the start to make this process as easy as possible. In order to achieve this, use should be made of intelligent hardware using microprocessor. By doing this it is possible to place at the hardware level a degree of fault detection that up to now has been unattainable. By correct design, it should prove possible to detect faults down to a very small module level, although it may be argued that, due to the miniaturisation of components, this level need not be that small, giving a corresponding reduction in the amount and complexity of the fault detection circuitry.

If the impression has been given in preceding paragraphs that all hardware faults are well-defined and permanent, then a few words of caution are necessary. While the majority of hardware faults are indeed of this type, there are enough of the obscure and transient faults in the hardware to make practical detection extremely difficult.

**SOFTWARE CHECKING.** At present in DISCUS most of the checking is done in software. This provides for bounds checking on all the arrays in global store and the access type (read-only etc). This checking is a part of the operating system and not inherent in the language used to write the operating system. The operating system can only check those faults that pass through it, it cannot check applications errors. These errors would have to be catered for by the designer of the function software. There are however facilities in the operating system that the applications programmer can use to help the function recover from an error.

As we will see the present aids to checking and recovery are somewhat limited - especially in the hardware. A Mark 1.5 version of DISCUS has been designed and will be the described later in this section. Before I do that it is necessary to look at the features a little more generally, and also their application in a future design of a DISCUS like system.

## 4.4 MEMORY AND I/O ACCESSING

This covers several areas most of which are normally done in the operating system or inherently as part of the language used (PASCAL etc). It can be split into several areas -

**Priviledged Access.** This is where only certain parts of the system can access some of the memory or I/O ports. For instance, within DISCUS it is proposed that only the operating system be allowed access to the system I/O ports, and that the user be given only part of the I/O port range for applications use. This will prevent the user from deliberate or accidental use of the system level facilities. Most important of these is the AAR, since it is by this means each local processor gains access to the global store. As has been described above the Mark 1 DISCUS uses the inherent store partitioning provided by global store overlaying the applications programs to give some protection.

**Restricted Access of Store.** In PASCAL, if a program accesses an array with an array index outside the bounds for which the array was declared, then an error will be detected at run time. The GEC compiler for CORAL 66, with which the DISCUS O/S and applications programs were written, does not have this checking and if an array bound is exceeded, then no indication of this is given except for indeterminate program faults. However to help with protecting the global objects DISCUS contains bounds checking on all its global arrays (and local copies).

If we take PASCAL as example the following "program" should cause a run-time fault (providing that the compiler has the appropriate run-time checks):

```
VAR a:  ARRAY [0:10] OF integer;
VAR index:  integer;
FOR index:= 0 TO 20 DO
BEGIN
  a[index]:= index
END;
```

The feature is not a part of the definition of PASCAL but a function of the compiler - the CORAL 66 that we use at present could have this feature. As can readily be appreciated, this is very powerful for tracing faulty programs and for preventing inadvertant overwriting of data. However, a penalty must be paid for all this checking; more code and time. The amounts of extra code and time vary but they can be quite substantial. If it were possible to do this checking somewhere else, a considerable code saving could be achieved while still retaining the principle of bound checking. It is possible to order the compiler to produce code that does not include all these checks to increase speed when the program is thought to be correct and not need these checks. But it is impossible to say with any confidence when software is fault free. However if this checking was in the hardware a considerable amount of time and code could be saved.

The principle of "base-limit-access" checking in the hardware is by no means a new one (cf. the "capability" structure in Plessey's System 250 computer) and there are the "Memory Management Unit" devices produced by Zilog, Motorola et al. The main problems of this approach is deciding where it should go and how it should be used.

Before discussing the location and use of this facility, I think it is worth reviewing exactly what we want it to check and do. The first job must be to ensure that the current memory address being output to either local or global store is correctly within certain limits (hence "base-limit"), and that the operation about to be perpetrated on that address is valid (hence "access").

In order to provide a check it will be necessary to have registers that can be pre-filled with the parameters of the impending operations. Next we need some comparison logic to ensure that the address or action is within those limits set in the registers. If this process occured in parallel to the operation (i.e. the logic was in the Bus Supervisor Card) then the fault could only be detected and not prevented. The operation must not happen if the access etc is wrong. Because of this the checking must be done serially. However we would pay the penalty of speed since there was yet another unit to go through before reaching the destination. In the proposed P896 bus [27] it is intended that there should be some form of parallel bus supervisor. This device would monitor the operations on the bus and be able to take over/modify a bus cycle if it felt it was necessary. This is both complicated to implement and restricts the bus speed considerably.

For this reason I think it is essential that the logic goes in between the master and the object of its access. If we assume that only the global store is to be protected by this means, then the only place for it is in the path to global store (on the Crate Bus Interface Card at present). Within DISCUS at present this card will prevent any Read-Only accessed store being written to. The source of the information is, however, the processor card (in the AAR), and if a hardware bounds checking scheme were used, then that part of the AAR would have to be on the Crate Bus Interface. This scheme would prevent any illegal transaction to the global store from being made.

It would be essential that this mechanism only be accessible from the operating system otherwise it would be possible to alter the access registers from the users program, thus letting a user access global store illegally. As can be appreciated there is a penalty from having this checking both in the hardware and software. In the hardware it is estimated that a further 20 integrated circuits might be used on the Crate Interface Card to make a custom MMU, and the commercial devices would be difficult to apply to DISCUS in its present form. Also there would be a penalty of access time to the global store, although this would probably make very little difference in terms of overall system speed. In the operating system software, the registers would have to be set (via I/O instructions) before every (or group of) global store access(es). However the software time involved would be less than that incurred to check the access validity entirely in the processor's code, as indicated in the case of PASCAL above.
An assumption was made above of only having the checking in the global store.

The Crate Interface Card is able to check all transactions with other processors and prevent a single local processor from corrupting another's data area in global store. At present, DISCUS is able to do this only in its operating system, provided that global store accesses are made by the operating system and not the user program. If we now include the entire local store as well as global store to be checked, a further difficulty arises as to where this checking should be carried out. At first sight the most obvious place would seem to be the processor. All transactions to store, both Local and Global, can be checked before they leave the processor card. However, this is all right if the processor is the only master module on that local bus. If, as is used in DISCUS, there is another master card (i.e. the peripheral card) that card would be able to corrupt local store unless it had some form of register checking also.

From this, it could be thought that it would be best to place the registers actually on the memory cards themselves. However, the disadvantage of this would be resetting the registers to a new value for a different master module. Some form of stackable registers would be needed, and the complication of this would not be worth trying against the previous method.

If we were to use a Memory Management Unit we can spilt the store into a number of segments (in commercial units these are usually from 256 bytes up to 64 Kbytes, with a total address space of up to 8 Mbytes). It is also possible to use several MMUs to increase this figure. Each segment has associated with it a register I will call the DESCRIPTOR register (in the Plessey System 250 computer these are equivalent to the CAPABILITY REGISTERS). This descriptor register contains a BASE ADDRESS field, a LIMIT field and an ATTRIBUTE field. For the observant I am using the ZILOG Z8010 as an example of a typical MMU.

**Base/Limit Field.** The MMU takes a logical address and transforms it into a physical address that is required to access the memory. This process is called RELOCATION. This means, in a system where the various modules of the entire processor are not on a single card but on a backplane bus, that this bus will be carrying local addresses if the MMUs are placed on the store card rather than a CPU card. (This is considered in a little more detail below.) This relocation should be entirely transparent to the applications programmer. Each of the base address registers in the MMU associated with one of the segment addresses via the TRANSLATION TABLE. The logical address is added to this base address to form a complete physical address. Since the MMU usually divides the memory into consecutive 256 byte blocks at smallest, the lowest 8 bits of the address are common to both the logical and the physical address. The limit register contains a value N that says that N+1 blocks of 256 bytes have been allocated to that particular segment. If the processor tries to access out of this limit, it will cause a violation error (see below).

**Attribute Register.** The attribute registers control the type of access to the segment to which the Segment Description Register refers. The access encompasses both DMA and CPU activity. Typically the register might consist of the following -

i. **READ ONLY.** This prevents a write command from being made to the segment. In the current version of DISCUS this would be equivalent to using the WPMS line. Within DISCUS however, this attribute only refers to store on the global bus.

ii. **SYSTEM MODE.** On most of the newer 16-bit microprocessors there are priviledged instructions that can only be executed in the so called SYSTEM mode (see section on MICROPROCESSORS FOR DISCUS). The equivalent within DISCUS is the use of forbidden areas for the operating system and global store (see below on DISCUS 1.5).

iii. **CPU Inhibit.** This flag allows only a DMA module, not the CPU, to access a segment. This permits external peripherals etc, to have access through the system in a protected fashion.

iv. **Execute Only.** This restricts access to the segment to only instruction fetch cycles. Within DISCUS at the moment, it is possible to run programs that are resident in the global store, but this is strongly not recommended since unpredictable results may arise if the AAR is changed as the program is being run. In the next DISCUS it is intended that this will not be possible by using a very basic mechanism that cannot be changed. The only place for the MMU would be on the memory card rather than the gateway down to global store where the global protection mechanism would be.

v. **DMA Inhibit.** This allows only the CPU to access that segment. The value of this can be appreciated when it is realised that within a system there will undoubtedly be intelligent peripheral cards under the applications designers control. By using the access capability of this card, it would be possible to corrupt any part of memory without this DMA inhibit. In the new DISCUS system, some method of only going to areas preselected by the operating system would be highly desirable.

vi. **Direction and Warning.** This checks for potential segment overflow and is particularly useful for stack operations.

vii. **Changed.** This flag indicates that a particular segment has had a write command. This is useful in checking whether an array has been written to. Within DISCUS it would be used with the array objects. A function could check on this flag to ensure the array had not been interfered with.

viii. **Referenced.** This is similar to the changed flag, but it refers to both read and write.

All these facilities would be available and used by the operating system in protecting the data and code from deliberate or accidental misuse by the applications programmer. Generally the faults that are being guarded against are software errors not hardware. Checks on the data such as parity are considered separately below.

The MMU is accessed via protected/privileged instructions that are only available in the system mode. When an applications program is run, it is done in the normal mode. However, not all microprocessors have this facility, and it would be necessary to place the MMU control registers in an address range that cannot be accessed by the applications programmer, much as it is intended in DISCUS 1.5. In order to use successfully this type of MMU, it is necessary to make one or 2 assumptions about what the bus must carry. The primary information, apart from Data/Address and basic read/write controls, must be status/type of the command being executed. This encompasses such things as Instruction Fetch cycles and stack operations. The latter is not so relevant with many microprocessors since there is no definable stack "per se", only another area of store accessed by one of the internal CPU registers. Information such as memory refresh is not needed, since the DISCUS system is intended to be totally asynchronous and each of the memory cards will have its own independent refresh system. Also, the global store will not have a CPU to generate these requests, although there will be some from the bus controller. Having to have microprocessor internal operations available does however drive us towards a single card local processor, with a component level bus for any future machines.

## 4.5 BUS FAULT DETECTION

There are many faults in a system that can be avoided by good design and operation of the bus. Up to now, there have been very few designs of bus that have given any attention to this area. A great number of faults can and should be trapped at the bus level and the two types are detailed below.

**Handshaking.** It is essential that all actions within a computer can be acknowledged so that the device doing the controlling has some idea of the success of its command. It is also essential to have some scheme as the *addressed slave device* may not be ready and would need some method of causing the master device to wait. However, the problem that occurs here is one of time: how long an interval is deemed to elapse before a slave device is considered to be absent or to have faulted. Within DISCUS, an arbitrary time interval was chosen that was considered to be well in excess of what would normally be acceptable from the various modules within DISCUS. The time interval for local accesses is approximately 5000 times the normal access time required. Since this is not required except in fault conditions, it was thought to be acceptable.

All the commands on the DISCUS bus are handshaked and are:

1. DMA Commands
2. Global Store Requests
3. All I/O Memory Commands

These signals all have a single level of acknowledgment, but it might be better to have several levels with one instruction. In other words, to take as an example, a memory read instruction, the following could occur:

1. Master Issues the Address

    2. Slave acknowledges valid address

3. Master Issues Memory Read Command

    4. Slave acknowledges command

    5. Slave gives back data

6. Master acknowledges data

The advantage of this method is that when a fault occurs in the system, it is easier to trace from the above 3 levels that just the one level. However care has to be taken in the protocols in case any extra time is taken on the bus to carry out these actions.

When a bus fault does occur, it is up to a checking mechanism on each local bus to detect this and act accordingly. The most convenient way of doing this is by the use of a time-out. There is a device that monitors the bus and when there has been no reply to a command for a specified time it causes some remedial action to take place. Usually all it can do is signal to the operating system that a fault has occurred, and tell it what type of command was being done and where it happened. It must also give the handshake to complete the protocols. I will return to this mechanism later when we look at the Bus Supervisor Card.

**Data Integrity.** Any corruption of a single bit leads to the corruption of the whole, and it is thus essential that the correct address or data arrives at its destination. The simplest and most convenient way of achieving this is by adding PARITY bits to the bus.method. It is a well tried method and does not need any further discussion here except to say that for a system such as DISCUS, it would probably only be necessary to have one parity for the address and one for the data. The circuit requirements for implementing a single bit parity scheme would probably amount to about 8 integrated circuits. This would provide checking and generation.

It would be possible to use some form of error correcting code on each of the address and data busses. This would enable recovery action to be carried out directly by the address/data logic, a scheme which has its attractions since it is distributing the recovery action throughout the system. However, the mechanism must not just act without telling the appropriate monitoring software that it has made some correction. This is because the fault might be symptomatic of a larger problem that will need to be corrected. The main disadvantage of this method is that of cost. Until large-scale integrated circuits are produced that contain all the necessary logic, this method will use too many integrated circuits and use up too much board space to make it worthwhile.

There are other places in the system where parity checking etc, can be used more conveniently. It is probably in a system such as DISCUS that more errors will be produced by the memory devices used, among these it is probably the RAM ics that will be at fault (especially the dynamic RAMs). As the geometry of the individual cells in the RAMs gets smaller, so they are going to be more and more susceptible to pattern sensitivity and nuclear effects such as alpha particle emission by the substrate material. Because of this, it would be desirable to include extra checking bits in the RAM array. In DISCUS it is proposed only to use a single even parity bit for every 8 bits of RAM.

RAM is easier to parity check than ROM since the former is made up of 16K x 1 bit devices giving bytes stored in parallel across as many chips as there are bits in the word and one addition of a single parity bit is like the addition of another bit in the word. With ROM the devices currently available are organised as byte parallel usually as nk x 8: in DISCUS, the standard 2k x 8 bits are used. Adding a parity bit for ROM would need the addition of another complex ROM, and generating it each time a new ROM was produced would be very time consuming and tedious. It is unlikely that much benefit could be gained from having parity in the ROM since non-erasable ROMs rely on a physical change in the device to store their information and are thus resistant to such faults indicated above for RAM stores.

## 4.6 FAULT DIAGNOSIS

Now that we have looked at some of the various methods of actually detecting the fault when it has occured, it is necessary to consider the much more complex area of diagnosing the fault, if possible in more detail than that indicated by the fault detection circuitry. It may be necessary to invoke active methods to carry out this analysis by having a separate processor module on each local bus. Unlike fault detection, the isolation of faults must necessarily involve the operating system. However, at present it is intended only to discuss the hardware aspects.

The first thing to discuss is to what level should one be able to diagnose a system. The object of the diagnosis is to replace the faulty unit (either manually or under machine control) so that the entire system can resume working at its normal capacity. To achieve this aim it is necessary to replace a particular component part of the system. This part may be a single electronic component or it may be the entire system. Both extremes are clearly not practical in what we are considering, so it is necessary, as part of deciding what recovery action is necessary in a system to discover what is the most convenient size of module to replace. From this can be gauged the type and amount of recovery. Ideally we try to give the diagnosis mechanism the maximum amount of information regarding the fault. For a hardware fault we need status registers that store the state of the machine when the fault occured. The operating system is then able to inspect these at leisure to make a suitable judgement on what it thinks happened, and thus what it should do to try and get things correct again.

In order to look at the types of fault detection etc, I am going to use both the present DISCUS and the proposed Mark 1.5 facilities as an example.

## 4.7 PRESENT RECOVERY IN DISCUS

Before we look at the various mechanisms for detecting faults, it is an important principle to realise. If the system is designed inherently to prevent abuse then the detection mechanisms can be correspondingly easier. One important feature of DISCUS is the principle of overlaying the applications code on the global store.

As we saw in section 3 it is impossible for any code that is in the top half of local store to access directly the global store. The action of accessing global store from the top half of local store causes the top half to vanish. Thus the applications code that was running there effectively vanishes. We can make use of this by putting the operating system in the lower half of global store and the applications code in the top half. Thus the applications programs have to go through the operating system in the lower half of store to get at the various global objects. However it is still possible to corrupt global store if a program written in the lower half and then it was run. The way to stop this is to ensure that the applications code cannot access the lower half of the local store, only the operating system can do this. As well as this the system I/O ports for the AAR and the fault interrupts have to be protected by similar means. I will return to this when we look at the "ark 1.5 DISCUS.

he recovery aids are very simple because when DISCUS was first conceived, it was impossible to tell what was going to be required on the final machine. Now that experience has been gained on the types of problem that arise, a better understanding of the recovery aids has been gained. Although not all the aids that will be discussed are going to be put on the current DISCUS, it is hoped to try out some of them on a Mark 1.5 to enhance the machine that has been described. However a quick review of what is currently done on DISCUS before continuing is necessary.

The only method that the hardware currently has of detecting a fault is by the command acknowledge time-out expiring. This indicates that one of the 4 main commands on the bus (MERD/, MEWR/, IORD/, and IOWR/) has not been acknowledged. At the time of the error it is not known whether the slave being addressed has faulted, or whether it is even present. In the latter case we have the further complication of not knowing whether it was the calling program in the master that has been corrupted and is issuing the wrong address to non existent memory, or whether the store or I/O card has not been included in the hardware (in the early stages this is quite a common occurrence). Also, there may be a write to a write protected device, either from the write protect in the AAR being set or by trying to write to ROM.

At present another complication is that it is not known which master is doing the accessing; for instance, it could be the main local processor or the VDU peripheral card. It is the Bus Supervisor Card that detects bus fail errors by having a time-out on the command activity. Currently this card is passive and is only able to carry out very primitive actions. These actions can be summarised as:

i. It can ignore the failure completely, thus allowing the whole system to freeze at the point of the fault. This allows a manual inspection of the bus to be carried out. It also issues a fault signal to the local front panel card to light the appropriate fault light. Of the three options I have found this to be the least used.

ii. It can provide a default handshake as soon as the error has been detected, and like i. it gives the fault light. This allows the system to continue operation. In the initial stages I found this the one to use.

iii. As ii., but with a selectable interrupt to the processor so that a fault routine may be entered if necessary. This interrupt is to one of the 8 interrupt levels on the DISCUS bus to the processor (although if the processor itself has faulted, this action becomes a waste of time and user has to provide assistance). At present this causes a simple subroutine instruction "CALL address" to be given to the processor forcing the processor to enter an interrupt routing. This means that the processor saves the current instruction pointer onto the stack for an eventual subroutine return. This operation is most important since the address saved is the one at which the fault occurred. Although it may not be the faulty address, since the operation that faulted might have been an indirect memory operation or I/O instruction. It is up to the operating system to analyse the state of the register and the instruction that was being acted upon at the fault time.

As may be appreciated this all gives very limited help to the operating system for diagnosis. It does not at present even give the address that faulted only the address of the instruction that caused the fault. It would be necessary to analyse the instruction to see where the fault might be. It would be possible to save all the registers within the cpu to help with this, but it would be difficult to do. Thus the operating system can only register the fault to the operator, throw away the present transaction, and wait for another transaction. It does not give the operator any clue as to why the fault occured or where it occured.

A further complication is the length of the default time-out. If made too short there are occasions where a perfectly valid operation may take many times the time-out - for instance, when loading or initialising the global store. If made too long the system will noticeably pause when a fault occurs. With the original system the default time-out was disabled so that the processor paused indefinitely when the fault occured. This was not very satisfactory, and the Mark 1.5 attempted to cure this.

What I have explained above are essentially hardware diagnostics - when a store module goes wrong etc. In the present DISCUS the recovery actions are a trifle brutal - throw away what we were doing and wait for a new stimulus. This means that we nearly always get to a consistent state, but that we lose one complete transaction. In the telephone exchange type of application this may not matter since the user can always dial again.

We have a problem if the fault was permanent, rather than a single transient one. A permanent fault will keep recurring and the system will keep failing. Some form of counter to count the faults is needed. Only after a certain number of faults will the operator be called.

There are basically two methods of alerting the operator. First we can have a display device on every processor that the operating system can use to write system messages to, (since functions can be duplicated every processor must have a display). Second we could have a special centralised error reporting function with only VDU in the system. Although having a central "control" function might be desirable, we chose to have one display per processor.

## 4.8 DISCUS MARK 1.5

The Mark 1.5 is an attempt to put some of the ideas above into practice without having to completely redesign the whole system. I will look at the system in a little more detail to show how much was achieved towards recovery. As we saw above there has to be some form of "serial" or "in-line" detection of potential accessing faults. I showed that the master modules were probably the best place to put these detection circuits. Since the Mark 1.5 is basically a Mark 1 DISCUS with some new boards and a minimum of change at a system level, some retreat from this is necessary. Only the processor card has this detection circuit. The following is a list of the constraints on a final Mark 1.5 system.

i. All other master modules besides the main local processor would be constrained to access only a discrete part of store. At present in DISCUS, the peripheral card is capable of addressing all of the Local Store at locations fixed at a code level when the peripheral card software was designed. With the new card these areas of store are defined by the main system and can be dynamic. The peripheral card is supplied with the address of these buffers. There is no check however on the integrity of the peripheral card code. However, it is still unable to access the global store.

ii. The operating system should be a fixed part of store. In DISCUS at present, this is the first 16k of local memory. Even with it written in a high level language, this should prove ample - currently the operating system is about 10 kbytes written in CORAL 66.

iii. As it stands, the applications programs are all confined to the top half of the local DISCUS store. This gives 32 kbytes of store. If this is not considered enough, then perhaps the point of distributed software has been missed and the user had better try an improved functional split of the software.

iv. There are a restricted number of I/O ports available to the applications programmer. These are ports 00H - 0AFH, the others are reserved for system use. I feel that this number is sufficient for most applications.

v. There should be no executable code in global store - only data. In the early stages of development, the programs for each processor are loaded via the global store from a disk-handler DISCUS processor, but they are not executable from a local processor. At present this is not the case with DISCUS, it is possible that a processor can run programs anywhere in global store. A word of caution is necessary here; if the AAR is set in error to a different value while running a global program, unpredictable results will arise, and a multi-processor system like DISCUS is complicated enough without introducing that sort of hazard.

The new system consists of three new cards:

i. 8085 Processor Card
ii. Z80 Peripheral Card (VDU etc)
iii. Intelligent Bus Supervisor Card

Only the first two are concerned with recovery so I will deal with them alone, and only those parts which are concerned with recovery.

### 4.8.1 8085 Processor Card

As its name suggests, the 8085 processor card uses the enhanced version of the 8080. This device removes a substantial amount of the external circuitry that was needed to drive the 8080. As a result more circuitry can be fitted on the card to provide the extra facilities. The 8085 can go much faster although at present the system is only about 30% faster. The instruction set is identical, except for the addition of two instructions to use the extra interrupt facilities. This means that it runs the same object code that the CORAL-66 produces. The operating system will have to be changed to use the new facilities, but the applications programs will remain the same. Only regenerating the system will have to done.

I will briefly discuss each new facility and how it is intended to help the system and the operating system in particular.

**Timers.** There are three counters on the 8085 card to provide the operating system with an accurate real time clock. The basic pulse to each counter is 1 microsecond. Each counter can interrupt the cpu so that real time clocks and watch dog timers could be used. All the software necessary to generate software timing loops can be removed, only the interrupts for the real time clocks are needed.

**Access restriction.** As I have said elsewhere it would be desirable to check each command leaving the processor card and ensure that it is a valid operation. In the present system a complete capability structure is not feasible without a vast amount of change and extra circuitry. However a basic level of program protection has been added. A set of restrictions on how store and I/O space is to be used has been produced.

i. The operating system shall always be resident in the bottom 16k of the local store. The operating system shall be able to access anything throughout DISCUS in both local and global store.

ii. The applications programs shall be in the upper half of the local store (32k - 64k). They shall be unable to access anything in the lower 16k of local store where the operating system is. They are also unable to access directly global store since the top half of local store overlays the global store. I/O ports between 0B0H and 0FFH are forbidden.

iii. Programs in the remaining area (16k - 32k) shall also be unable to access store and I/O in a similar fashion to the applications programs.

To sum up the above, only the operating system is allowed to access the system bits. This means that the applications programmer cannot access the global store directly. As we saw above the present DISCUS can be defeated by the applications programmer writing a program in the lower half of global store and then running this program to access the global store. The applications programmer could in theory access the Auxiliary Address Register directly from the top half of store, but it will not do him much good except crash the system. Now both these situations are impossible.

If the applications program does try these accesses we have to do two things: firstly we have to stop the access from taking place. This ensures that he does not do anything harmful. It is a matter of hardware logic to achieve this and is not available to the applications program.

Secondly we have to alert the operating system. The latter can then endeavour to take some remedial action. The mechanism to alert the operating system must be invisible to the applications program. For this reason the interrupt that is used must be non-maskable, and is why the TRAP interrupt is used. Once a fault has been detected there has to be some way that the operating system can try to diagnose the fault. The easiest way is to have a status register on the card that is updated every time a access fault occurs. (This means that only the last fault is seen, although this is not a hardship). I have found that only six bits are needed to give the operating system a fair chance.

The various access conditions detected by the status register are:

i. Memory write to illegal area
ii. Memory read to illegal area
iii. I/O write to illegal area
iv. I/O read to illegal area

There are one or two more conditions that are held, but are not relevant to the operating system. What the operating system does when a fault occurs is in two parts. A warning message is given to operator, and then some remedial action to try and recover the situation. A message is sent to the Bus Supervisor Card which displays it on a small display. How this display is arranged is explained in more detail below.

### 4.8.2 Bus Supervisor Card

The present Bus Supervisor Card is essential a passive "bus-sniffer". It provides a way of looking at the bus via an umbilical cable. It provides the bus termination, bus arbitration, and most importantly it checks for bus fail timeouts. The new card does all these things but several more key activities.

i. It can read the bus when a time-out failure has occurred and produce some form of message on the display.

ii. It can inhibit the time-out mechanism when required. For instance when the main loader is working (see the Appendix on the loader [29]).

iii. It can increase the time-out so that accesses to global store can be catered for.

iv. It can give a better indication to the operating system for diagnosis.

v. It can display messages from the local DISCUS processor.

vi. It provides the local normal reset to its DISCUS processor. The BOOT local reset has been removed. The only BOOT reset is the overall one that comes from the global store crate and resets the whole DISCUS multiprocessor.

The display that is used is a simple 16 character alphanumeric display that is mounted on a front panel close to the processor it serves. This display removes the need for having a VDU on each processor for displaying error messages - an expensive and bulky activity. The operating system sends all the abort messages to the display.

These take the form:

```
        "ABORT - xx      "
```

Where "xx" is the appropriate error number. When an access fault occurs, the following type of message will be sent:

```
        "SYSTEM TRAP - xx"
```

In this case "xx" is derived from the 8085 status register described above. When a bus fail occurs the following type of message will be output:

```
        "BFAIL   aaa-xxxx"
```

Where "aaa" is the access that failed and "xxxx" is the address at which it failed. For instance:

```
        "BFAIL   MRD-0012"   memory read fail at 12H
   or   "BFAIL   IWR-  B2"   I/O write to port 0B2H
```

The only problem at the moment is to decide on how several fault indications at once should be dealt with. Some form of store is needed to nest fault messages. The display is controlled (as is the whole card) by a special microprocessor. This is the Universal Peripheral Interface (UPI). This device is treated by the local DISCUS processor as a pair of I/O ports that can be addressed. It can inspect the DISCUS bus and control the various functions of the Supervisor Card such as the timeouts. Because the UPI is addressed via restricted access ports only the operating system can use it.

We have not fully discovered its possibilities (or hazards) since at the time of writing only a prototype has been tried. However the results are very promising, even though the operating system has not been fully changed. The ability to output "comfort" messages to the operator without a VDU is a useful one.

Although the entire system has not been fully run in all possible ways with the new version of the operating system, I feel sufficient has been learnt to go towards a completely new machine and what we might need. What I have described above is probably the most realistic changes that can be made to DISCUS without making it a mess of hardware cabling and add-on circuits, and it is definitely not intended to add any further facilities or change anything else to the present DISCUS.

## 4.9 MORAL

It is important to realise that what I have described above is only the detection part of the recovery process. At present we are still experimenting with recovery methods and hope to produce some more detailed views later. It is impossible to cover all the aspects of the software and hardware here so I will give a short reminder of what we have discussed above. I will return to some more of these points in the conclusion along with some of the software problems in designing for recovery.

i. Recovery must be considered from the start, it is not some facility that can be easily added on later.

ii. As much information as possible should be given to the recovery mechanism by the fault detector. It is better to have too much than too little.

iii. The hardware protocols should be enough to do all the operations between modules, but simple enough to deduce what has happened.

iv. At some stage an operator will have to come and sort out the mess; so make sure enough information is given to him.

## 5. MICROPROCESSORS FOR DISCUS

It is an important concept of DISCUS that it should use commonly obtainable components. Nowhere is this more important than in the choice of the microprocessor and all its associated systems. There are many reasons for choosing the microprocessor and must take into account every aspect of the final DISCUS system. Many of the more obvious criteria like the basic word length of the microprocessor are not as important as the system criteria - such as the choice of the development system.

**THOUGHTS ON THE NEW MICROS.** At the time of writing there is a new generation of micros being proposed. These are the APX range of microprocessors produced by INTEL. These are very sophisticated VLSI microprocessors with a range of tightly coupled peripheral chips. For applications which can use the standard development system with the manufacturer supplied software these micros will have the power of a machine like the PDP-11/34. The processors have an integral operation system with co-processors on a component level bus sharing the specialised functions such a floating-point operations. Because of this it will be increasingly difficult to adapt them to the DISCUS type of system where research on different methods of hardware configuration are being tried. What can be done with the micros will depend wholly on the software and operating system supplied by the manufacturer. One might speculate that the current range of micros such as the 68000, Z8000 and 8086 are the last devices that can be used in such user specialised applications such as DISCUS. For these reasons and that there has not been sufficient information to make a realistic discussion on the APX range, we will restrict ourselves with the present 16-bit machines (68000 etc).

**SUPPORT CHIPS.** Most microprocessors today are not produced in isolation: rather they are surrounded by a range of specialised devices to carry out functions that the microprocessor has not the room for (physically on the die), or that it cannot do very efficiently. These devices range from the simple I/O peripheral controller (for example the 8255 from Intel where the number of pins is the restricting factor) to such devices as the various floating point arithmetic chips and floppy disk controllers. Very often specialised applications define which support chips should be used and thus which microprocessor rather than the other way around. With the current DISCUS a variety of support chips are used from a variety of manufacturers. All the devices are interfaced to a standard component level/system level bus. If it is required to make use of a specialised device the compiler, and thus the operating system must be written in such a way so as to use this device most efficiently or else some of the point of using it will be lost. This will then make the software highly hardware dependent which may not be desirable if different hardware configurations are being tried. Care must be taken in the use of these devices so that one is not restricted at some future date.

### 5.1 HARDWARE CONSIDERATIONS

Not every microprocessor is suitable for selection as a candidate for inclusion in an array of processors. In fact it is fair to say that nearly all the microprocessors produced to date have to be made to work by having special logic to cope with the various needs of the multiprocessor environment. The exceptions to this rule include the F100, Z8000, MC68000 and 8086. The latter 3 are recent additions and are able to cope with an extended address range (more than 1 Mbyte) and include some of the more advanced features of multiprocess working such as memory mapping and dynamically relocatable code.

There are several features that a microprocessor must possess, or be made to possess, if it is to be used in DISCUS. These include:

i. Command acknowledge
ii. Suspend operation facility
iii. Indivisible store "test and set" commands

### 5.1.1 Command Acknowledge

It is essential in any asynchronous system that the processor or commanding module is synchronised with the slave it is using. The most convenient way of doing this is for the slave to provide an acknowledgement signal to indicate to the master that the latter's command has been recognised at its destination. This can give several advantages for the system:

i. The system can cope with a variety of different module speeds. There are a variety of different store media presently in existence and it is essential that they are able to delay the master, if it is to ensure that valid data is returned.

ii. It is possible to include a very basic level of fault reporting by virtue of the non-return of the acknowledge signal from the addressed module. This can be caused in 2 ways. Firstly the module could have faulted and be failing to give the response, or the program, due to a fault at either applications or operating system level, could be addressing a non-existant module. Both of these conditions should be detectable and preferably separately identifiable.

In order to get the maximum benefit for satisfactorily detecting and isolating possible errors at a bus level, it is necessary to go to a multi-level handshake system. For instance, the address and the *command* issued by the master can each have their own quite separate handshake lines. Thus any particular command cycle of the master proceeds in a series of checkable steps. The address is issued and if not acknowledged, then the module is either not there or the address recognition mechanism has failed. Assuming the master receives the correct acknowledge an appropriate read/write command can be issued. If this is not acknowledged then the fault could be either failure of the command mechanism or else a "write to read only store" type command. As can be seen this dual-handshake mechanism can provide immediately an enhanced capability for first level fault detection. Although probably not desirable for engineering reasons, it would be possible to further extend the number of acknowledge lines to each different type of command (read, write, read-modify-write etc) to give even more detail at this level of faults.

### 5.1.2 Suspend Operation Facility

It is common practice, even on simple multiprocessor systems with separate local store, to provide for direct memory access (DMA) facilities. This allows for several master modules to co-exist on the same bus accessing common local slaves. There are 2 classes of control for this.

i. Each master claims time on the bus for the common resources on an equal basis including the primary control processor. This implies that there must be an overhead of time, albeit a small one, in the instruction cycle of each processor. Generally there must be a separate bus controller providing either asynchronous and synchronous control of the bus utilisation requests. An example of this bus is the INTEL Multibus, as used in the MDS-800 minicomputer.

ii. If the DMA requests are few, then it is possible to allow the master processor to always assume that it has control of the bus. In order that the processor can be suspended, some form of HOLD facility must be provided. The mechanism of this is very straight forward. When the HOLD command is issued to the processor, it completes its current instruction cycle and then instead of issuing the new program instructions address, goes into an indefinite wait state. In addition the address, data and command lines are released to a high impedance state thus allowing the requesting potential master module to take control of the bus. In order for this to work properly, there must be an acknowledge to the HOLD, which is issued from the processor to confirm that it has suspended normal operation and has released the bus. This facility could be implemented at either microprocessor or board level within the system.

As well as a HOLD facility there must be a mechanism for preventing the suspension during special instructions or sequences of instruction. This occurs during test and set instructions.

### 5.1.3 Indivisible Store Commands

There are occasions in multi-processor working when certain areas of common store are accessed by a processor with that processor knowing that it has the undivided control of that store. This is used almost exclusively to provide semaphores/flags for the software control of data stores or message stores. To explain the necessity of this facility consider the following. If there is a data area, Fig 16, whose total contents must be kept consistent to one another (or to put it more colloquially, when one entry is changed all the entries have to be changed) it is vitally necessary that the reading processor (#2) does not read the data while the other processor (#1) is still writing into it since processor 2 will be reading invalid/inconsistent information.

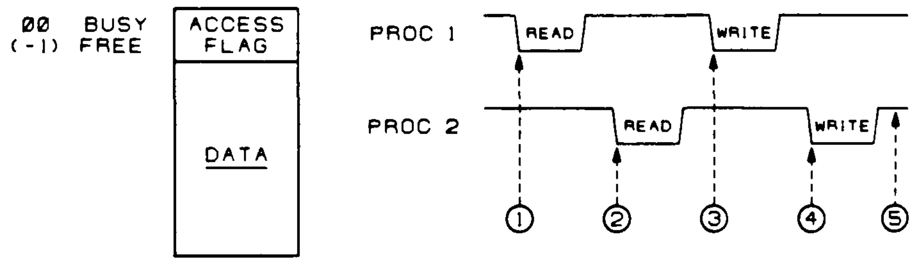

**FIG 16.** **Shared Store Contention**

Using the bus activity diagram (Fig 16) it is possible to see how this state can arise.

i. Processor 1 reads the flag and finds it to be -1 (free).

ii. Processor 2 also reads and finds the data area free.

iii. One instruction later, Processor 1, thinking it now has the undivided attention of the common data writes the busy flag, and then proceeds to alter the entries.

iv. Processor 2 also having read that the data is free, starts to read, not realising that it has not complete control of the data.

v. From this point, chaos can quickly become evident. However, there is a danger that because of only slight discrepancies in the recovered data, the system is unable to recognise that a fault has occurred, until some time has passed and suitable corrective or recovery measures become virtually impossible to carry out successfully.

When 3 or more processors are involved in addressing the data area, the possibilities of disaster become legion. It is worth noting here that the use of simple busy/free flag is not completely suitable for more than 2 processors, and it is necessary to qualify the flag with, say, a unique processor identity to denote who has the store. The mechanism of how this is achieved in DISCUS is explained more fully in the description of the operating system [8]. In order to stop this type of error occurring, there must be a mechanism to allow a processor to make indivisible operations on the store.

There are 2 basic ways of achieving this:

i. **Test and Set Operations** (Read-Modify-Write). There is a special instruction that keeps the address bus valid through a number of separate read write control cycles. This means that there must be a unique instruction called "Test and Set" that can be used so that the flag read/writes are not separated by an extraneous instruction to fetch, thus losing the continued valid flag address. By having the address valid and separate commands during this period means that there must be 2 levels of handshake to cater for the address and commands.

A typical operation would be as follows:

| | Buffer Free | | | Buffer Busy | |
|---|---|---|---|---|---|
| | Flag | Acc | | Flag | Acc |
| Start | -1 | x | | 0 | x |
| Read | -1 | -1 | or | 0 | 0 |
| Increment | -1 | 0 | | 0 | 1 |
| Write | 0 | 0 | | 1 | 1 |

Free = -1 Busy = anything else

In the case where the buffer is free, the processor has booked the store during the indivisible part and is able to see afterwards whether it has got access at its leisure knowing that if it has control it has automatically booked it, but if it has not got control the store is marked as busy. The only problem here is that after many processors have been trying to access, the busy flag is considerably more than zero, and a real danger in 8 bit micros is that byte overflow occurs and the busy flag becomes zero again, thus invalidating the mechanism. However, software protection can be included to overcome this [8]. It is necessary when releasing the buffer therefore that decrement store is not used and that -1 is written in directly to clear the flag. At the time of writing, only 2 microprocessors are known to have the facility of read/modify/write: the Ferranti F-100 and the Motorola 68000, both 16 bit machines.

ii. **Separate Memory Local Command.** Where a special read/modify/write command is unavailable, it is necessary to generate a separate bus command or method of holding a common store address valid for more than one instruction. It has been usual up to now to employ the former method. The Intel 8086 microprocessor has a separate command "LOCK" which holds a memory locked bus line true for one whole instruction cycle (usually increment memory). The command is invoked by using an instruction prefix in the assembly code thus:

```
LOCK INR M
```

DISCUS uses a similar method, but the Memory Locked Line is generated from a special on-card input port line, thus allowing several instructions to be carried out on locked store.

The danger with this system is when the locked line gets stuck during a fault, either in hardware or software, it prevents any other processor from accessing all or part of store (see below on store partitioning).

### 5.1.4 Other criterea

There are several other considerations in the hardware that we must consider when choosing a microprocessor, and I will look at these briefly below.

**WORD LENGTH.** At present there are really only two choices for this: 8-bits or 16-bits. Within a machine like DISCUS there is no need for any floating point arithmetic as standard so the value of a long word for precision is irrelevant. The only benefit for DISCUS would be an increase in bandwidth when moving data throughout the system. The word length should not be a significant reason for the choice of a microprocessor - it should only be the final consideration when two microprocessors of similar standard, yet differing word lengths, are being considered.

**PROCESSOR SPEED.** Like the word length the speed is a desirable feature, not a necessary one. Most microprocessors today have a basic instruction cycle of about 1 microsecond or less with the shortest being about 500 nanoseconds. This should prove adequate for most DISCUS applications.

**ADDRESS RANGE.** It has been found in the current applications of DISCUS (a small communications switch) that the standard 64k addressing range is adequate for local store. The amount of global store is not nearly so critical at 7 blocks of 32k. In order to test the handling of large data-bases within DISCUS I wrote a multi-user "fantasy adventure" game [28] which used nearly all the global store but I feel this type of application is very much the exception. (In partial defence of this seemingly trivial application, the program which is by no means trivial to write, identified several faults in the system generator and operating system that would have otherwise been undetected.) Future applications may use large data bases - especially in areas of network emulation where individual network nodes are being emulated by individual DISCUS processors. It is however an arguable point that functions should always use small amounts of local store when mapped onto DISCUS, bearing in mind the "brain sized" package argument for the software. One of the advantages of the present DISCUS addressing scheme is its inherent protection of the global store from the applications programs by using the overlay method. That is where the global store blocks are overlaid with the top half of the local DISCUS store (the top 32k) where the applications programs are placed. Thus these programs cannot access global store directly either accidentally or deliberately. The 16-bit micros have a linear memory rather than this overlay system, thus any natural protection afforded by the DISCUS overlay system is lost. It would be tiresome to arrange for the newer microprocessors to have a similar arrangement to the present DISCUS. However there are a variety of special chips that can provide a measure of protection to the store (see elsewhere on the Memory Management Chip produced by the Zilog Corporation - Z8010). The new DISCUS operating system would have to be rewritten quite drastically to cope with this different approach. Until it is investigated more fully it is impossible to see all the implications of changing to a fully linear store with some form of memory management device.

## 5.2 WHICH MICROPROCESSOR?

I feel that the choice of the microprocessor for a Mark 2 DISCUS is between two devices - the Motorola 68000 and the ZILOG Z8000. Before looking at the various capabilities of each of these, I will look briefly at the others. It is certain that I am going to annoy some sacred cows on the way, but I offer no apologies for so doing, the discussion is biased and ultimately represents my personal choice, based on much experience.

### 5.2.1 INTEL 8086

This was the third 16-bit microprocessor to be produced. It was the first of the third generation devices to arrive however and is perhaps the strongest of the runners up to the new DISCUS micro. It is backed by the considerable expertise of INTEL, who offer a very comprehensive range of support chips for it. INTEL have produced a micro that is aimed primarily at upgrading the 8080/8085 family that is used in DISCUS at present. It is able to use all the 8080 type peripheral chips with little trouble. The internal architecture reflects the compatibility with the 8080, but because of this it suffers from many drawbacks in the use of the internal registers. For instance, it is not very symmetrical - it is impossible to use any register as the stack pointer or program counter. Otherwise the instruction set is fairly comprehensive and allows a variety of data manipulations and string handling. One of its biggest drawbacks, and the reason I have not considered it, is its lack of reserved or priviledged instructions. As I have said elsewhere, it is essential for a secure machine that the applications programmer be kept away from the system. A good way of doing this is by reserved instructions. The 8086 will access up to 1 Mbyte of store and 16 kbytes of I/O ports. There is a segmenting scheme that uses four base registers. These provide for data, stack and code segments with an extra segment. There are registers for indexing in string operations. The 8086 relies on special registers rather than general purpose ones which I think is a disadvantage.

The 8086 has CORAL, PASCAL and PLM-86 for it. At the time of writing, the latter two are supplied by Intel, and the former by two British companies. As in most of its other products Intel's software is very good, and a minimum of trouble could be expected from using the device. However it is limited by its poor architecture and not having priviledged instructions.

### 5.2.2 TEXAS 9900

This device is relatively old and slow. Texas were the second to produce a 16 bit micro. It can only address 64 kbytes of memory and has relatively poor support with peripheral chips compared to the other 16 bits micros. It has one big advantage over the others by having all its registers in RAM. Thus it is easy to change context by merely changing a pointer - an advantage in processing interrupts etc. It is sufficiently cruder than the others for me to discard it out of hand with no further comment.

### 5.2.3 NATIONAL SEMICONDUCTOR 16000 Series

I must confess that I know very little about this micro. It can address 16 Mbytes with provision for virtual memory operation (it says in the sales literature). There are 8 32 bit registers for general use. It has supervisor/user modes and separate stack pointers for this. The instruction set is comprehensive and supports several operation modes. These include "memory relative" and "scaled indexed". The former provides two levels of indirection on the operand. The latter provides the facility to index an operand and can be combined with all the other addressing modes. It is particularly useful for "RECORD" structures in high level languages. I do not know what languages National intend to support, but I am sure there will ones like PASCAL. The development facilities are somewhat unknown, and I intend to ignore the 16000 series because of insufficient information.

### 5.2.4 FERRANTI F100

This was the first 16 bit micro to be produced. It is still the only full military specification micro in production. It is slow compared to the other devices and tends to be used in military applications, with the result that very few reviews of 16 bit microprocessors mention it. It can only address 32k bytes and has few internal registers. The range of support devices is small. There is a peripheral chip that can be configured for a variety of tasks from bus buffering to managing multiprocessor busses. It has CORAL as its high level language. The development system provide for the usual facilities with disk storage, assembler and compiler. Again, like the 9900, it is now sufficently out of date for me to dismiss it from the discussion.

As a final comment on the above it is impossible to be wholly unbiased, ultimately the choice is personal preference and what you feel happy using (and understanding). The choice is based on experience and what you think will do the job in the best way. Having narrowed the choice to two devices it would be better to look at the remaining devices in a little more detail. I apologise in advance if the following looks like the manufacturers' data sheets, but it is unavoidable.

### 5.2.5 MOTOROLA 68000

**Address range.** The 68000 can address 16 Mbyte by itself, and 64 Mbytes with external support. It does not use I/O space as such, everything is memory mapped, but with 16 Mbytes of direct space available it does not really matter.

**Internal registers.** There are 18 32-bit registers. Seven are address registers, eight are data registers, with the remaining being stack registers and program counters. The data registers can be used as bytes, words or long words. The address registers do not support byte operands. The instructions that move the registers allow practically all types of moving and loading between the registers and memory. There are an adequate number of registers for providing virtual program counters, and virtual stack pointers for languages such as PASCAL.

**Addressing modes.** There are 11 addressing modes providing virtually all the access types that might be required.

**Data Types.** For a DISCUS like machine the full range of data types is not generally needed. BCD arithmetic has never been used on the present DISCUS, these specialised facilities in the assembly language are a little difficult to make use of in the compilers. However the 68000 can support a number of types -

Bits - fairly useful when controlling external systems.
2-digit BCD
8, 16 or 32 logicals, signed and unsigned integers

**Procedure Call Support.** When a high level language is considered this area is important. How such things as recursive routines are implemented on the machine can make a considerable difference to the size and complexity of the final code. The 68000 has the following instruction types to do procedure calls -

Simple procedure calls with stack
Save/restore registers on stack
Link/unlink stack
Call, return, load and push effective address operations
Move instructions combined with auto incr/decr addressing modes

These are fairly normal microprocessor/computer instructions. However there is one pair that is worth looking at - LINK and UNLINK. These two allow very easy manipulation of what is known as the "environment" or sometimes the "stack frame" in high languages. When a procedure is entered a certain amount of store is needed for the local variables. If these variables are assigned to absolute locations in store when the program is compiled it will prevent the procedure from calling itself. This is know as recursion and is most important. The reason is that the storage locations that are fixed will only contain the last level of call. When the procedure returns to itself these locations will no longer contain the variables in the first level.

For this reason for a procedure to be recursive it must have a separate local data area for each call of the procedure. The most convenient way of doing this is to grab some store from the stack. This stack may not necessarily be the same stack as is used for storing the return addresses. Each time the procedure is called a pointer is given to the procedure that defines the base of an area of store that is used by the procedure. Variables within the procedure are addressed as offsets within this area from the given base. How this pointer is allocated is immaterial in this case. When the procedure returns the old pointer must be restored to the calling procedure, so that the previous variables are available.

In order that this can be done an amount of stack and pointer shuffling is carried out. On most microprocessors this involves several instructions. With the 68000 there are two instructions that enable these operations to carried out with only one instruction for procedure calling and one for returning.

The first is the link stack instructions. The instruction

```
LINK  An,displacement
```

will allocate an area of store for the procedure call of size "displacement" and adjust the frame pointer in address register "An" and the system stack pointer accordingly. In algebraic or "RTL" notation this action is -

```
[sp] := An
sp := sp - 2
An := sp
sp := sp + displacement
```

All local variables are based on the new value of the frame pointer in register An.

The unlink instruction is even simpler. The instruction

```
UNLK  An
```

will reclaim the data area and restore the old frame pointer. This action is -

```
sp := An
sp := sp + 2
An := [sp]
```

This is one of the most useful features of the 68000, and one that is not done on the Z8000.

**System/Normal working.** The 68000 has two working states - normal and supervisor. The supervisor mode handles all the exception processing such as bus failures and divide by zero. There are some instructions that can only be used in the supervisor state.

STOP  -  of dubious use
RESET  -  useful for external devices, or under error conditions
RTE   (Return from exception)
MOVE to SR   (Move to status register)
AND (word) immediate to SR
EOR (word) immediate to SR
OR (word) immediate to SR
MOVE USP  (Move to/from user stack pointer)

The supervisor state is used for the handling of exceptions. Exceptions are situations that require processing that is not in the normal course of the program. The 68000 has the following types of exceptions -

Reset
Bus errors
Address errors
Trace
Interrupts
Illegal instructions (i.e. non existant or unimplemented ones)
Priviledged instructions in normal mode
TRAP, TRAPV and CHK (check against bounds)  instructions
Divide by zero

I do not intend to discuss any of these in great detail, the Motorola data book should be consulted for this. However as a general point the "non-normal" state is the supervisor state, rather than just a system state. Being in the supervisor state usually implies that something has gone wrong. It is not a state that an operating system can use as its normal state.

**String Handling primitives.** String primitives are commands that allow simple operations to be carried out on character strings. For instance - compare two strings in given buffers. The searching will stop either when a difference is found or the end of buffer is reached, usually detected by a register being preset the length of the string and then decremented. The flags then reflect the outcome of the operation. Another use is in string processing languages such as LISP, but DISCUS is unlikely to make use of this. The 68000 has no string primitives. However if a high level language is being used, it will require a very clever one to detect when these primitives could be used. In normal use they will be probably be only used when assembly code programs are being written. I do not consider their presence or otherwise as very noteworthy when trying to decide which micro to use.

**Arithmetic error traps.** In the 68000 these detect the conditions overflow and divide by zero. If arithmetic is being used then these are useful, but not essential.

**External Support Devices.** The 68000 will be able to support the following types (and I stress that at the time of writing with some of them it is only "will").

Direct Memory Accessing. Within DISCUS this is not terribly important as a special scheme is used. Also by the nature of having one processor to one function the only place that DMA could be used with the 68000 devices is on a local processor card.

Serial Interfaces. VDU type interfaces are very important. If on the new DISCUS the processor is made on one card containing the micro and all the interfaces, it is possible to use a component level bus. This allows 68000 compatible chips to be used properly. The 68000 will support V24 type, HDLC and IEEE 488 type protocols. Of these the latter will be least used in DISCUS unless for communication to external peripherals.

Memory Management Units. These are very important for the next DISCUS as has been explained elsewhere. The 68000 MMU will have 32 variable sized segments for each MMU . Each segment will have a "capability" register associated with it to provide base/limit access and write protection of segments. Although the data is somewhat sketchy so far it unlikely to be too different to the Zilog MMU already described above.

**Multi-processor facilities.** This requires the ability to "lock" store somehow to enable semaphores to be safely set without interruption for other masters. As we saw above this is most important for multiprocessing. The 68000 has the indivisible test and set instruction

```
TAS <effective address>
```

TAS uses the read/modify/write instruction cycle of the 68000 and is inherent in the hardware.

**Development Facilities.** Motorola provide development facilities on the following machines -

M6800 EXORciser Development System
M6809 EXORciser Development System
IBM 370 Systems
PDP-11

There are simulators and assemblers on all the machines. The simulator on the EXORciser machines include hardware emulation. Motorola intend to support several high level languages for the 68000 include PASCAL, FORTRAN and PLM (a version of PL/1). Probably BASIC will also be provided. It is inconceivable that other software manufacturers will not implement other languages on the 68000. Prime candidates for this will be APL, ALGOL-68, COBOL and ADA.

**Miscellanea.** Other points about the 68000 that are worth noting are in no particular order of importance:

i. It has non-multiplexed data and address lines. Consequently it must use a large i.c. package (64 pins). The trend on modern busses is now towards multiplexed data lines, so that there would have to be some external conversion circuitry. This might not be as bad as it seems since buffer would have to be supplied and these two functions could undoubtably be combined.

ii. Single shot. This is done by forcing an exception, which allows a debugging/monitor program to monitor the execution of the program under test. The facility is turned on and off by a flag in the flag register. It would be possible to only trace through a specified portion of the code. Many extol the virtues of having single shot, but I can see no redeeming feature if a high level language is to be used. A proper approach to debugging at that level is necessary. Having to single shot through a compiled version of the original HLL seems to be tedious to the extreme. Worse than that it might encourage people to correct their programs by patching the machine code - this is vile. The next DISCUS will not even contemplate using single shot. It is worth noting also that the present DISCUS was developed entirely without using single shot that I provided for the local 8080 cpu card. In fact I removed it from the 8085 Mark 1.5 processor card. I am sure I will be accused of missing the point about this entirely - so be it, my opinion still stands.

iii. There are sufficient status lines output to allow the external circuitry to track the internal operation of the 68000.

Generally the 68000 is a good device, which is probably the fastest of the 16 bit machines. It has a good architecture for compiler generated code, and a reasonably compact one for assembly code programming. The range of support chips should be good. The documentation is very comprehensive also.

## 5.2.6 ZILOG Z8000

**Address range.** The Z8000 can address 24 Mbyte of user space and 24 Mbyte of system space by itself. It can also have 16 Mbyte for each Z8010 Memory Management Unit. In parallel with this there are 65 kbytes of I/O address space. Zilog support two versions of the Z8000: the Z8001 which has the full addressing range using segment bits in a 48 pin package, and the Z8002 in a 40 pin which is just 64 kbytes without the segment bits.

**Internal registers.** There are 15 general purpose 16 bit registers. These can configured as 8, 16, 32 or 64 bit registers. Like the 68000 the load instructions allow most types of movements between registers and memory, although I feel they are not quite as "symmetrical" as the 68000. Also like the 68000 there are an adequate number of registers for providing virtual program counters, and virtual stack pointers. There are two stack pointers in registers 14/15 that provide for the system and the user. Only register 0 is special in the instructions and it cannot be used in all accessing modes - not very convenient.

**Addressing modes.** There are 10 addressing modes which like the 68000 providing virtually all the access types that might be required.

**Data Types.** The Z8000 can support a number of types -

Bits
2-digit BCD
8 or 16 logicals
8, 16 or 32 signed integers
byte strings
word strings

Again 2 digit BCD is virtually useless, although bits are useful. The byte and word string instructions are of dubious value since it would take a very clever compiler to use them effectively.

**Procedure Call Support.** The Z8000 has the following instruction types to do procedure calls -

Simple procedure calls with stack
Save/restore registers on stack
Call, return, load and push effective address operations
Push and pop registers
System calls

Compared with the 68000 these are not quite so comprehensive. The one key omission is an equivalent to LINK and UNLINK. While these can undoubtably be done from two or three others instructions, it is a pity that it does not have them. The System Call makes up for this lack more than adequately in my opinion for a DISCUS type machine, and we will have a look at this below.

**System/Normal Working.** The Z8000 has two working states - normal and system. The system mode handles all the exception processing such as interrupts and segmentation traps from the MMU. The system state has a large number of reserved instructions that can only be run when in system mode. These instructions are -

HALT - again not terribly useful
Interrupt processing
All input/output commands
Load control register
Load program status
Multi-processor instructions

As I indicated when looking at the 68000 the system state in the Z8000 is a little different from the 68000. Not only does it provide for the exception processing, but it also provides a way of running priviledged instructions. These instructions can be used by an operating system to control accesses by the applications program which runs in normal mode. Any attempt to overcome this mechanism by the user program will result in a exception. The Z8000 has the following types of exceptions -

Interrupts
Illegal instructions (i.e. non-existant or non-implemented ones)
System instructions in normal mode
Segment traps

Again I do not intend to discuss the above exceptions in great detail, the Zilog data book should be consulted for this. There exists one instruction that is most important for implementing system calls from the user program that must be looked at. There has to be a method that the latter can use to enter system mode. At first sight this seems to require the user to have control over the system/normal flag, which of course violates the point of the system mode. Also the applications program cannot write or call the system area without a violation. A simple call to the operating system procedure will only run the that procedure, <u>but in the normal mode</u>. This will cause a violation the first time a system instruction is done. There is an instruction system call -

```
SC  n
```

This will call the system procedure n and automatically go into system mode. The number n is a vector to the operating system procedure needed. When the instruction is run the number n is placed on the stack and is available to the operating system. The SC instruction always vectors to the same place predefined in the program status area at compile time. The initial system procedure then extracts n from the stack and then jumps to the appropriate operating system program.

**String Handling primitives.** The Z8000 has a rich set of string primitives. However as we saw before if a high level language is being used, these are unnecessary for most DISCUS applications, so I will ignore them.

**Arithmetic error traps.** There are no arithmetic error traps in the Z8000.

**External Support Devices.** The Z8000 can support the following types:

**Direct Memory Accessing.** There is a fairly standard DMA unit.

**Serial Interfaces.** The Z8000 has a combined V24 type and SDLC/HDLC type interface. It has NRZ/RZ and FM data encoding so it is about as comprehensive as might be necessary.

**Memory Management Units.** Having discussed this device in great detail elsewhere I refer you to that section above. For any who know this device I have already discussed it in all but name in the section on recovery so little more needs to be said here.

**FIFO Interface Unit.** This unit is one of the most useful that Zilog produce. It is a method of joining two asynchronous data streams. It provides all the handshaking necessary to ensure correct operation of each system. It is 128 bytes deep and eight bits wide. It is possible to cascade and parallel the devices to increase the width and depth of the FIFO. The FIFO is very important in a DISCUS like system that relies on being totally asynchronous.

**Error checking devices.** There is a "Burst error processor" that can provide error detection and correction for any data stream. It can work up to 20 Mbits effective data rate. A variety of polynomials and correction methods are supported. If DISCUS is to be concerned with integrity of data these devices will be essential to protect data on the transfers in the channels and to and from global objects. The new 64k RAm chips will require some from of error correction locally also.

**Multi-processor facilities.** The Z8000 does not have instruction "test and set" like the 68000 "TAS". Instead there are two pins on the device - multi-micro in and multi-micro out. These pins can be manipulated by four system level instructions -

| | | |
|---|---|---|
| MBIT | - | Test multi-micro bit input |
| MREQ | - | Multi-micro request |
| MRES | - | Multi-micro reset |
| MSET | - | Multi-micro set |

The multi-micro pins are daisy chained from one potential master to another giving a geographical type precedence. In small multi systems or ZILOG "specials" this is fine and all the above instructions are used to provide the basis for indivisible instructions. It has the advantage that any instruction can be made indivisible - sometimes very useful. In the DISCUS like machine the multi-micro output pin can be used as a Lock indicator as the present DISCUS system uses. It can be used to generate an externally done "read-modify-write" memory cycle. The instructions MRES and MSET can be used at minimum to control this facility. However with a little extra circuitry the daisy chain mechanism could be converted to a parallel scheme.

**Development Facilities.** Zilog provide development facilities on the following machines -

PDS 8000/20A Z8000 Development System
PDP 11/44, 11/45 and 11/70
ZLAB 8000

There are simulators and assemblers on all the machines. The development facilities for the DEC PDP series are worth noting. These are all written for the UNIX system. In fact since they are written in the C language they should run on any UNIX system.

The ZLAB 8000 is Zilog's new development system which supports up to 16 users on the UNIX system. It compiles several high level languages: PASCAL, C, PLZ/SYS. No doubt languages such as FORTRAN and BASIC will also be provided. The facilities provided by UNIX such as language parsers and lexical analysers would be well worth considering for DISCUS if special pre-processing software has to be written. The extension of CORAL for the present DISCUS was a major part of the system generator.

## 5.3 WHICH TO CHOOSE?

As a straight choice of micro the Z8000 is not as fast as the 68000, and the latter also as a more structured instruction set and internal architecture. If we were choosing a microprocessor for a high performance single processor system it would be the 68000. However we are not choosing one for this: facilities for data integrity, as we have seen everywhere in this report, are the prime consideration. The Z8000 is better for this type of operation.

Ultimately it is a personal choice that is coloured by experience. Experience with the manufacturer, experience with previous multi-processors, and experience with the languages that are offered. I have said elsewhere that the internal architecture and speed can be low priority reasons for choosing a microprocessor. The software, the development systems, the external chips, and those special features that are needed for the application in mind, are the prime considerations. It is possible that a new device will appear, or a new facet of the above two, that will cause me to review the choice - time will tell: but for the moment the Z8000 seems to fulfil our requirements best.

## 6. BUSSES FOR DISCUS

The bus is one of the most important items in DISCUS. If the processors are fragmented over several cards it will be used to carry all the data for both the local processor and the global store. Even if the system is made of truly single board processors it will carry all the global data. The present DISCUS uses a bus that, in all honesty, did not have much thought in the design, it just grew. I intend to look at what DISCUS might need now and what is available for it.

First, I think that it is relevant to ask the question - what information should go on the bus? This may seem like a strange question, but there are several types (or levels) of bus that exist within a computer system. I can summarise these levels as:

- i. System level bus
- ii. Backplane bus
- iii. Component level bus
- iv. Serial bus (V24, RS232 etc)

The **Component Level Bus** is the simplest to consider so I will deal with first. This should always be confined to a single card and should never be taken off that card. It connects the various devices on the card together and thus is processor or device (such as memory) dependent. The length of the bus is usually very short making line termination unnecessary. Except when going off card, buffering is usually unnecessary except when extra drive is needed for going from MOS to TTL. In many bus configurations today, it is considered that this component level bus will connect the basic elements of each processor together on one card: for example on Intel's SBC range of computer. By its very nature each processor card that has a different micro-processor on it will have a different component level bus on it. For this reason we need not concern ourselves with standards for this type.

The difference between the **Backplane Bus** and the **System Bus** is more difficult. Some people consider that a system bus should not carry instructions and that it should be totally asynchronous. Only data to and from some global store, or between processors should be carried on it. If each processor in an array was totally self contained then this would be feasible. However the processor might be split over several cards. It would desirable that this bus that contains the instruction stream is the same as the global bus. Thus the Backplane Bus would be identical to the System Bus as it is in the present DISCUS.

For this reason it is not intended that any distinction be made between them. The backplane/system bus is the only one that we are considering: the component level bus will be different for each card and it is impossible to set any form of standard.

When DISCUS was first conceived there were really no standard microprocessor busses that we could use. The only candidate was produced by INTEL: The Intel MULTIBUS [30]. However this bus was unsuitable for what we wished to do and it is worth looking at it briefly to see why.

The MULTIBUS bus was originally designed by the INTEL Corporation for use in their MDS-800 range of microprocessor development system. It has since become a standard used by INTEL in their SBC range of single board computers and by others manufacturers supplying boards of that range. It is now such a popular industry standard that the IEEE has produced a standard specification (IEEE 796). The DISCUS loader uses an MDS system designed around the bus with a special interface card from the Multibus to the DISCUS global store.

The bus can support a variety of modules that can transfer data at up to 10 MHz rate. There are a set of protocols that provide for both daisy chain and parallel arbitration for Master/Slave working on the bus. The bus allows several masters to be resident on the bus for multiprocessor working.

The bus was originally designed for the 8080 microprocessor and the choice of control signals reflects this. Undoubtedly the MULTIBUS may be suitable for the INTEL micros but it is doubtful whether others (Z8000 and 68000) could easily use the bus - I stress the word "easily". There is a large amount of redundancy in the control signals with separate signals for memory read/write and input-output read/write. The bus arbitration is synchronous with a bus clock providing the necessary reference. There are 16 data lines and 20 address lines (with the future possibility of 24). There is no form of error checking on these lines. Like the S100 bus there are 8 interrupt lines working on a one out of eight basis which could be reduced to 4 lines.

The bus is accessed by 2 dissimilar direct connectors, one of 86-way on a 0.156" pitch and a second of 60-way at 0.1" pitch. At the time I wrote this the latter has no formal definition and forms no part of the specification. This should not be bussed on the backplane so that it can be used by various manufacturers for special custom connections. In my opinion having a user definable area on the bus is definitely not desirable. It allows non-standard cards to be produced, which have no chance of being position independent on the bus. If geographical precedence was being used cards could not be changed on the bus to change their priority. Also if a card was inserted wrongly the possiblity of having a card full of dead bus interface chips on the subsidiary connector would be fairly high. I think that the design of this part of the system should be standardised as soon as possible.

The reasons why I did not select it for DISCUS (bearing in mind that the bus was selected about 1976) can be summarised as:

i. There was no facility to lock the bus or provide indivisible operations.

ii. There were only 16 address bits.

A detailed description of the bus that was finally used for the present DISCUS can be found in the APPENDICES document. I am not going to dwell on it here except for occasional examples. I would like to consider what the DISCUS bus should consist of if I were to start again, and then see what is currently available.

## 6.1 TOWARDS A NEW DISCUS BUS

First it is important to list what the general requirements for such a bus are. The following is a brief list that is based on work that is being done on generating a new microprocessor bus for the IEEE (P896). I have been involved with these through the IEE, so they represent current thinking on the subject.

i. Support Multiple Module Architecture. I have specifically not said "Multiple Processor". Having the latter phrase implies that there are potentially only a fixed number of processors within the array. It might be required that there should be a single card (module) with several processors on it. There are two reasons for limiting the number of modules that can become masters on the bus. First one of physical size: the bus can only be a certain length long in order to ensure correct electrical performance of the bus. Secondly the bus arbiter needs a set of arbitration lines to make each card electrically unique for the arbitration protocols. The processor identity need not be the same as this arbitration level.

By saying that the bus must support a number of separate masters, we imply several other features of the bus. The protocols for claiming time on the bus must be capable of handling several requests. These requests will be asynchronous in order to cope with a variety of different masters.

ii. Reliable/recoverable and redundant. There is no doubt that the majority of computers being designed today put this as one of their most important criteria. Any bus that is considered must not detract from these aims. The area is fairly broad and covers:

   a. Parity protection of data. But not CRC type checks on blocks of data that are passed across the bus; these are system defined mechanisms, not functions of the bus.

   b. Protocols that lend themselves to easy diagnosis where it is obvious that they have faulted and what is wrong with them. Thus the fewer the types of bus cycle that are needed the better. Strange exceptions to the normal protocols are only confusing and only make fault detection and diagnosis more difficult.

   c. Multiple (redundant) busses should be supported. The protocols should be done in such a way that any number of alternate busses could be used by a module. There must be no signals that imply the state of another bus.

iii. Independent of manufacturer, processor or architecture. The design of the bus should imply any particular scheme or device. Although we are constraining the architecture to be a bus-based one.

I intend to look at some of these areas in a little more detail below, together with some of the problems that the constraints will imply.

### 6.1.1 Synchronous versus Asynchronous

The first thing to decide is how the bus is going to operate in its basic transfer cycle - asynchronous or synchronous. It is necessary that any decision here is ruled by the behaviour of the global store. This bus may not have any form of controller and in a tightly connected star configuration it will probably become an extension of the local processor bus. In this case the various global bus actions must proceed at a rate governed by the local bus. When they are tightly connected these 2 busses cannot be synchronous. A bus cycle on the global bus does not run in time with the local. There are 2 methods commonly used to achieve synchronous busses on both global and local stores. The first is to ensure that the whole multi-processor uses a common clock and synchronising line. The CYBA multi-micro processor built originally at Swansea University uses this method and many of the early problems were associated with providing a clean, skew-free clock. Because of this, the system does not lend itself to easy reconfiguration and addition of processors. It has the advantage that it is possible to repeat a run exactly (external applications hardware allowing) so that faults can be diagnosed more easily.

The second method is not wholly synchronous in that each of the various busses throughout the system work synchronously in themselves and the interface card provides a suitable buffer between the various busses. Obviously at some stage on this card there will be a point that is prone to asynchronous behaviour. The normal way to join the the two asynchronous paths is to have a first-in, first-out store (FIFO) to allow each bus to work at its own speed. However there must be some form of handshake to ensure that the correct information is passed to and from the global store.

The next DISCUS will almost certainly be asynchronous again. If there are no other ways of providing the arbitration, I feel that the present system is good enough. However there are new busses and methods being produced that may well be more suitable. The disadvantage of the present circuit is that it is star connected, and therefore a bit "heavy" on wires that do not easily fall on the bus. However it allows the interface to global store to be much more simple. If 2 or more paths to global store are used this method reduces the amount of hardware on this interface to a minimum. With each crossing of an interface the new system is synchronised to the bus requesting system. This allows for several interface boundaries to be crossed and for the destination bus to be used as an extension to the requesting system. There are several disadvantages to this. The first one is that the system can never be run twice with exactly the same results as can the synchronous system. This is only required for a small number of faults where it is not possible to provide some form of software synchronisation to allow process/function synchronisation (an entirely different subject). I do not think it to be a great disadvantage especially since DISCUS is probably going to be used with equipment which is asynchronous and never behaves identically on each run.

With a synchronous bus everything proceeds at the rate of the system clock so that the speed of the system is limited by this clock. An asynchronous system is limited only by the handshaking and the speed of the system accessed. This allows memory of a variety of speeds to be used without very much difficulty. The speed of the interfaces between each bus is limited by the response of the asynchronous arbiter. In practice I have found this to be an acceptable speed limitation and is nearly always the majority delay in accessing another bus (except for the gross delay of another processor having control of the bus).

DISCUS is limited by this controller to a star connected system rather than a "daisy-chain" system. By its nature the controller provides a rotating priority scheme which in DISCUS is considered to be an advantage since no one device can monopolise the bus (except if the bus is locked by the accessing device but this could be checked by the bus monitor card). For the present then we will assume that the bus will be completely asynchronous with no overall system clock.

### 6.1.2 Address Lines

It would obviously be unwise to limit ourselves to a number of lines that prevent expansion with future systems. It is not expected that it will be necessary to go above 32 bits which will give an address range of 4000 Mbytes. Thus the bus is capable of going up to this size, but only about 24 bits of address will be used.

If DISCUS is to be concerned with recovery then some form of data integrity is needed on the bus. For the address/data lines it is suggested that one odd parity bit per 8 information bits is used. Thus a maximum of 4 parity bits is needed for the full address range.

### 6.1.3 Input/Output

In the present DISCUS there is a separate address space of 255 bytes specifically for Input/Output using the 8080 I/O instructions - IN x, OUT x. Some microprocessors do not have these extra instructions (6800 etc) and rely on parts of the address range being mapped into I/O ports - hence the name "memory mapped I/O". This allows the full range of normal memory instructions to be applied to I/O ports. There is no reason with address range proposed (32 bits) why one of this bits should not be used to indicate a memory-mapped I/O range. This removes the need for the I/O or memory signal line. Also memory-mapped and I/O type microprocessors can be used on the bus with no redundancy.

### 6.1.4 Data lines

Like the address it would be unwise to restrict the choice to only 16 bits but leave room for expansion to 32 bits. The bus must be bidirectional (unlike the original S100 bus that had separate lines for read and write). Like the address lines there must be 4 parity lines associated with the data lines.

### 6.1.5 Byte/Word Accessing

The old microprocessors such as the 8080, Z80 and 6800 are all 8 bit word micros. The new ones use the bus in a 8 bit and a 16 bit fashion. In, say, the 8086 there is an extra address qualifier BHE/ that defines what the transaction on the bus is performing.

The following table indicates a general scheme that could be used -

| A0 | B/W | |
|---|---|---|
| 0 | 0 | low (even) byte to low 8 data bits |
| 0 | 1 | 16 bit parallel read/write |
| 1 | 0 | high (odd) byte to low 8 data bits |
| 1 | 1 | high (odd) byte to high 8 data bits |

This scheme allows both 16 and 8 bit processors to work on the same bus and both of them can access individual bytes. When the 8-bit processor accesses the bus the byte/word (B/W) line is left alone. On a negative true bus this means that this line will float via the terminating network to +5 V, which is logical zero. The normal addressing with the least significant address bit being used for the normal 8-bit address range.

There is however a problem with this - no one can agree whether the low byte is on odd or even addresses. Intel and the Digital Equipment Corporation consider that even addresses are low bytes, while Motorola and Zilog take the opposite scheme. The final design of the memory boards can be either way round and provided that a common scheme is used it will only depend on which microprocessor is used.

### 6.1.6 Data Storage

A subsidiary problem to the above is how addresses in different microprocessors are stored. Basically there are 2 methods:

i. Least significant byte of the address stored first and most significant byte last. (This is sometimes referred to as "Little-Endian", from the satire by Swift "Gullivers Travels".)

ii. Most significant byte of the address stored first and least significant byte last. (Sometimes referred to as "Big-Endian".)

The Intel 8080/8085 and the Zilog Z80 both use Scheme i. With the Motorola 6800 and 6502 using Scheme ii.

Advantages of method i.

a. When the least significant byte occurs first in a direct addressing instruction, adding bytes on the end causes an increase in significance. In other words bit zero is always in the first byte. This makes it easier for compilers to work with varying word lengths. When the least significant byte occurs last, the position of this byte is variable, making life more difficult for the compiler (see case 2).

b. When storing variables of differing word lengths it is not necessary to know the length (i.e. the number of bytes) that the variable occupies in order to coerce the variable to a shorter length if the bytes are stored in order of increasing significance. For example consider the following for a mnemonic assembler similar to the 8080.

   Accessing a byte with least significant bit first -

```
          LDA  FRED          ;Load byte direct
          LXI  H,FRED        ;Load byte indirect
          MOV  C,M
```

   Accessing a byte with most significant bit first -

```
          LDA  FRED+1        ;Load byte direct
          LXI  H,FRED+1      ;Load byte indirect
          MOV  C,M
```

   In all these cases the variable is stored as -

```
      FRED: DS   2
```

The key to the difference is the "+1" after the variable name. If a double precision variable of four bytes was used this would have to be "+3". This would make the compiler take special action for different length integers when coercing to bytes.

Advantage of method ii.

a. When addresses are put out in a memory dump they appear in the correct sense, although with a high level language the need for this is doubtful.

How character strings are stored does not really matter since the bytes are scanned from low to high address. The characters will appear in the correct sense when there is a memory dump.

Currently there seems to be a great leaning towards Method 2 for the new microprocessors. The reason for this is to allow the 16 bit microprocessor to fetch data in 16 bit words that has been stored as -

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
T h i s   i s   a   s  t  r  i  n  g
```

When method 1 is used the data would have to be stored as -

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
h T s i i   s   a s a  r  t  n  i  g
```

If a high level language was being used there should be no need for the programmer to worry how the data has been stored in memory. Debugging by patching in the machine code and inspecting how things are stored should be avoided at all costs, thus looking at memory directly should be discouraged. The only time that it would be necessary to know would be when peripheral devices write directly into the store for later use by a processor.

Generally I have no strong preference about which way the bytes are ordered within the memory or on the bus. If I had to make a choice it would be method 1 and thus least significant justification on the bus- although it must be said that there is very little difference between the two and method 2 is currently favoured.

### 6.1.7 Multiplexed or Non-multiplexed

In order to conserve pins on the bus it is sometimes arranged that the address and data are combined on the same lines. They are separated by time, and the correct synchronisation is achieved by having the appropriate control signals. While undoubtably releasing a great many lines for other control signals, the amount of extra logic required to produce non-multiplexed from multiplexed signals on each card is large. If we are to have room for 32 address lines, 32 data lines, 8 parity lines that leaves very little else for the control signals. It has been suggested that only in large systems are part of the data/address bus multiplexed. This allows for small systems to be made with non-multiplexed lines and have the bus extensions multiplexed. (See more of this below.)

DISCUS treats store interactions by a master-slave protocol at all times. This includes the accessing on the local bus. At present the local processor bus always has the processor card assume that the bus is permanently assigned to it so that it does not have to request time on its own bus. This saves a lot of complexity on the processor card, and since much of the activity on that bus is done by the processor card, and it saves any time spent in requesting the local bus. Any DMA activity on this bus will generally be in short bursts and can safely interrupt the processor card by using its HOLD facility to suspend operation. This allows a very simple interface to the local bus for both the processor and the DMA units (see the Appendices).

### 6.1.8 Multi-master Working

The ability for several master/slaves to share the same bus in order to control/access shared resources is a fundamental one. It can be very complex and it is up the bus designer to steer a course between this complexity where all possible eventualities are catered for (a difficult task since not all are foreseeable), and simplicity where only the minimum amount of lines are used on the bus; yet giving a sufficient amount of control that the system can be used in the majority of applications. It is important that not only the basic electrical and simple protocols of this control be studied, but also the implications that will occur at a system level. For instance the protocols necessary to support recovery mechanisms.

Before a module on the bus can acquire control of that bus to carry out transactions to other modules, it must go through some form of standardised protocol. This will result in others being kept off temporarily while it carries out those transactions. There are basically two functions that the decision or arbitration circuitry must carry out.

i. The primary task must be to ensure that only one master has control of the bus at any one time.

ii. A secondary task is to give and assert a priority to the requests so that every potential master will be able to get fair shares on the bus.

### 6.1.8.1 Ensuring Single Master

The basic form of this protocol is a single level handshake. The potential master gives out the signal "I want the bus", and then remains dormant until at some time later it gets the signal "You have the bus" from some outside source. When this latter signal is true then, and only then, may that master use the bus. When it has finished it removes the "I want the bus" signal indicating to the external controller/circuits that someone else can have a go.

**Asynchronous Problems.** Due to the asynchronous nature of these requests there must be some form of arbiter that can make sensible decisions in the event of two requests arrive simulataneously. If two (or more) requests arrive at the arbiter at precisely the same time there has to be some form of criterea that the arbiter can apply to decide who is to win control of the bus. This problem is caused by the inability of a simple RS latch to make up its mind when confronted by simultaneous signals. Thus most D-type latches do not behave properly when confronted by one of two strange situations.

i. When a very short pulse is presented to the reset of the latch it may not necessarily latch. It will never latch if this pulse is shorter than the combined propogation delays of the two cross coupled gates.

ii. If two inputs are presented to the latch at precisely the same time the latch is unable to decide which way to go. This "teetering on a mound of instability" can last for an appreciable length of time. The outputs of the latch can oscillate from one state to another before coming to rest.

Both these states cause flip-flops of whatever kind to exhibit what is known as the "meta-stable" state. That is a state which is not defined and cannot be said to have either of the two logic levels. This state can last for an appreciable time. With Schottky logic this state can last for hundreds of nanoseconds in the worst case, and during this time the external circuitry that the arbiter is driving must ignore this "meta-stable" state. Numerous papers have been produced [9],[10],[15],[16],[17] which discuss the problem and suggest what to do about it. It must be said that whatever these papers say the problem will never be cured - only made more improbable. The classic ways of avoiding the problem are to make the system synchronous or slow the system up so that the chance of "meta-stable" working is reduced. If the system is required to run very fast (greater than 20MHz request rate) it is really constrained to be synchronous. Circuits that arbitrate asynchronous requests always have a "flip-flop" in them. These may not be obvious, and may be over several gates. People who claim that they have found a complete solution are wrong - a careful look will reveal a familiar "flip-flop" somewhere in the circuit. It does not have to be a integrated circuit marked "D-type" etc. It must be borne in mind that there is no way of completely removing the meta-stable latch.

**Synchronous Working.** The simplest criterea that can be applied to an arbiter is that all the requests arrive at fixed points relative to the arbiters sampling time. This is usually achieved by making the requests arrive on one phase of the clock and sampling on the other phase of the clock. This removes any possibility of the requests changing at the sampling point. However there are two things that should be considered -

i. The clock must be provided to all the modules with minimum or no skew. This becomes troublesome when the propogation delay between the modules is an appreciable fraction of the half-period of the clock, and delays in the requesting circuitry might cause an edge to stray towards the sampling phase of the clock.

ii. The clock is a potential source of electrical noise on the bus and must be suitably terminated. Preferably it must also have its own earth return associated with it.

Although synchronous working does ensure that requests arrive a some fixed time, the problem of deciding between two requests still remains. This is usually done by giving each request an associated priority or identity. There are several methods of giving a priority to requesting input. These can be categorised as geographical and electrical.

### 6.1.8.2 Priority Schemes

**GEOGRAPHICAL PRIORITY.** The geographical priority refers to the position of the card within the bus having an inherent significance. The most usual of these schemes is the **DAISY CHAIN** or **SERIAL PRIORITY SCHEME.**

**Daisy Chain Scheme.** Each card has two lines "I've got" (RQ) and "I can't have" (GR/). Each line is connected to the card positions either side. In other words the RQ goes to the GR/ of the next card, and the GR/ comes from the RQ of the previous card. At the top of the chain the highest priority card has its GR/ line grounded so that it always has the bus when it asks, and by making its RQ line true it forces the GR/ line of the next card false thus forcing all the cards of lower priority to give up the bus (only one card should be using the bus) by a ripple affect. The lowest priority card has a RQ line that does not go anywhere and is totally dependent on the GR/ from the cards above remaining true.

Daisy Chain DMA is the system used on the Intel Multibus (IEEE 796 bus standard), and has been proposed for several other busses.

ADVANTAGES

i. It only uses two lines on each card with one line for the synchronising clock. Some systems use an extra line for Bus busy. This system is the most economical in the use of backplane connections.

ii. There is no central controller on the bus.

iii. There are potentially small amounts of control circuitry on each card.

DISADVANTAGES

i. It can take an appreciable amount of time for a request to ripple through from one end of the bus to another. This puts a limit on the speed of the system.

ii. The priority of a requestor is dependent on its physical position in the rack. A change of priority means a change of position. In some systems it is not necessary to change the position of the card, but just change some links on the backplane.

iii. It is possible for a low priority requestor to be shut out of the system for extended periods by continual use of the bus by higher priority devices. In some cases it would be possible for them to be shut out indefinitely.

iv. If a card is missing from the chain a jumper has to be placed in the chain.

**ELECTRICAL (PARALLEL) PRIORITIES.** If the position sensitivity and ripple effect of the daisy chain is considered an unacceptable constraint on the system (and in large systems it usually is), then some form of priority (PRx) associated with each request that is placed on the bus together with the request (RQ) must be used. There must also be a grant line to indicate that the requestor can use the bus (GR). The number of priority lines is determined by the maximum number of devices that are going to be needing the bus. The priority is encoded onto these lines in binary. Thus, assuming that there is no error mechanism (parity etc), there will be 5 lines (PR0 - PR4) for 32 requestors. Which level will be the highest will be determined by the design of the system. A major disadvantage of this system is that it is possible for a low priority requestor to be completely excluded from the bus for considerable lengths of time. The method of using a protest feature to allow low priority cards to acquire the bus is not altogether satisfactory since this line is really only another priority line (PR5 in the above example) and ultimately suffers from the same abuse. Another method is to have each card apply for the bus at some fixed low priority and then work its way up in priority every time it fails to acquire the bus. The disadvantage here is that at some time there will be two or more cards with the same priority level.

One advantage is that the system can be made with no clock and can be made asynchronous with a fair degree of reliability. Also there needs to be no central controller and cards can be missing from the system. Currently the best example of this type of DMA arbiter is the one that is proposed for the S100 bus (IEEE 696) and the new proposed micro standard in America - IEEE 896. This bus assumes that there is a central device (such as the main processor board) that needs to "held" in suspension while DMA activity is carried out, and is the focal point for the RQ and GR lines. This controller is still not totally resistant to asynchronous problems. It is possible for a priority to be asserted on the bus without its associated request. This is caused by an old friend. In the arbitration circuit currently proposed there are two D-type latches that provide the control signals for the priority and the request for the system. They are set by a pulse from the request circuitry. When I analysed the proposed circuit, I found that it was possible that this pulse could be shorter than the recommended latch parameters. Also since there are two latches the parameters will be different. Therefore one latch could be set and the other not. A priority could be set with no request and vice-versa. This would make the system "hang up" if the priority was sufficiently high that no other master could acquire the bus. It is worth noting here that the INTEL MULTIBUS requires that all the potential bus masters compete for the bus even the main processor.

ADVANTAGES

i. It only needs a relatively small number of bus lines. For instance 32 requestors with no protest feature can be accommodated onto 7 lines on the bus

ii. There need be no system clock to all the cards.

iii. The priority system need not be dependent on a cards position, only its identity/priority-level.

DISADVANTAGES

i. The circuitry is complex in order to get correct asynchronous working.

ii. It is still possible for a low priority requestor to be shut out from the bus. The protest feature that has been added is merely another level of request that can be overridden by another higher priority protestor. The system could lend itself to considerable abuse. Ideally there should be a protest line for each card - this then turns the whole scheme into the star connected system.

iii. There needs to be a something on the bus that gives back grants to the requests. If this is not the main processor (which assumes that it always has the bus except for the DMA requests) then the request/grant lines must be looped back on themselves. This is required when going to dumb global store as in some multiprocessors. If it is looped back then the tighter timing in the asynchronous arbiter may give rise to more metastable states.

**Star Connected Schemes.** There is a further method that is worth noting where there are no priority lines but each RQ and GR line is taken separately to a common point where the arbitration is carried out by a separate controller. This can be completely asynchronous from the requesting card but can be synchronised on the controller card. The only disadvantage is that that are a mass of unbussed wires leading to an unbussed card with special connections. I choose this system for the present DISCUS. There are a maximum of eight requestors on any one bus. By increasing the levels (or depth) of requests it is possible to get many more requestors (at least 32). With the new DISCUS it has been considered that there should be up to 24 processors in a single crate with its global store (the whole DISCUS in one crate therefore); this would give 48 lines on the back of the bus that were individually connected and not bussed. This is not very satisfactory for large systems.

Intel also use this scheme on their Multibus. The most common place is on the MDS-800 development system bus, where only every other slot has the connection to the parallel arbiter mounted at one end of the MDS-800 rack.

ADVANTAGES

i. Only two connections to the bus are needed for each requestor.

ii. There need be no synchronising clock on the bus.

iii. It is possible to design a rotating priority arbiter that will ensure that all requestors get serviced, and no one can dominate the bus. This is the scheme currently used on DISCUS and has proved to be satisfactory.

iv. In general the circuitry is very simple.

v. It would be very easy to design a controller that worked on a different priority scheme and one that was able to mask selected requestors. This circuitry is contained upon one card only.

DISADVANTAGES

i. There is lots of wires on the back plane that are not bussed. 24 requestors will require 48 extra lines on the back plane.

ii. There has to be one special card i the system to which all the request and grant lines are taken. This card is not part of the bus and would unless defined by a standard, be user defined (not desirable). It is worth noting that this card could contain a lot of other special circuits such as power supply monitoring, reset circuitry, and other "nasties" that need not go directly on the bus. If no other way exists around the bus starvation problem, then I would probably use this method on a future DISCUS, as giving me the most flexible control over the DMA commands on the bus.

### 6.1.9 Control Lines

With the amount of data/address/parity lines, the choice of control signals has to be very restricted and there should be no redundancy in the control lines. In other words there should be no signals on the bus that could be derived from any other signals.

Reviewing the requirements for such a bus:

i. It must be on one connector - so that a single card can use it for small systems and double Eurocard systems can use bus duplication. Also in a double connector if the second is not used for duplication it can be used for user defined I/O.

ii. It must be capable of taking up to 32 data and 32 address bits. These can be multiplexed or non-multiplexed where possible.

iii. There must be up to 8 error bits for data and address.

iv. Indirect connectors must be used.

v. The power connections should reflect an extension of the on-card power matrix.

In the following discussion I intend to take into account work being done on a new bus by the IEEE P896 committee in America. This committee was set up to produce a new standard for future microprocessor systems (the bus is sometimes called FUTUREBUS). The committee has been soliciting advice from engineers around America, Europe and the rest of the world. I have no intention in discussing the P896 bus in great detail here. However there are one or two points that are worth considering.

The P896 has a considerable problem fitting all the signals onto a single Euro connector - even the 96 way one. There are two levels that are provided. Level 1 is designed for the 64 way connector on the 96 way shell. It provides for a basic level using 32 bit multiplexed address and data. It has a parallel 5 bit wide bus arbitration scheme that allows up to 32 masters on a single bus. The bus is fully asynchronous and has a full set of read/write and acknowledge lines to carry out the command protocols. There is no I/O space only memory mapped space. At present the proposals have not been completed or published for comment, and the whole design remains somewhat fluid in definition. The level 2 bus contains the error checking functions and extra power connections. I hope that when the two levels are defined that level 2 will be defined completely first and then level 1 defined as a suitable subset that does not use all the facilities. Defining level 1 first and then trying to graft on a superset afterwards is not good design.

There are one or two points that I want to discuss in a little more detail as they bring up important points for DISCUS.

**Processor Control Register (PCR).** It is intended that there is a register on cards such as the processor card, that is addressed to control that card. It is a 16-bit register with each bit carrying a command such as RESET. This proposal is currently not to be used by P896 (at one time it was), but it is worth looking at none the less. While it seems to be a good way of reducing down the number of lines on the back-plane, I am concerned about its behaviour under certain circumstances. The initial system reset to the processor is my chief worry for two reasons.

i. (CATCH 22) - When cold starting the ability to access the PCR would depend on the processor being in a fit state to receive and honour a bus request from the card providing the reset. If it needs resetting at start-up it will not be able to give up the bus to the reset card in order to receive that reset. I feel it particularly dangerous that part of the system has to be "awake" at switch-on to arbitrate on the bus for PCR transfers.

ii. Often in a multiprocessor there is a central data store (as distinct from the local stores which have the code and local data) with a set of local processors in star connection around it. When giving a global reset to the local processors in a multiprocessor such as this it would be necessary to provide a controller on the global bus that can write to the local processors. It is an important principle for recovery that a local processor is unable to access another processor's code in the local store of that processor. This means that the control signals only flow one way - from a local processor to global store, not the other way around. This removes the natural partioning that is offered by a DISCUS like system, and thus this type of reset connection would be unacceptable.

I think it is essential for system security reasons that the initial reset is on a separate line on the bus.

The ability of addressing a particular card should not be a feature of the bus and contain such things as interrupts or DMA requests. Only specialised commands to cards should be carried out here. However it does allow the possibility of targetting commands to specific cards. Within DISCUS this would be particularly useful for recovery where it would be possible to interrogate cards for their status etc. There is one disadvantage however. When interrupts are used, usually by their very nature they are used in order to extract an immediate response from the device being interrupted. In the present DISCUS all the interrupts on the back/plane are used for system rather than applications use. Users who wish to have interrupt controlled peripherals should have a special card that contains the interrupt transactions to itself. This is done in a similar fashion to the intelligent peripheral card for DISCUS. Data is assembled and dealt with locally by this processor rather than the main local processor. The peripheral processor sends information to the local processor via DMA. Any interrupts on the bus would be at a system level and probably used for recovery mechanisms. Whether having an immediate response or not (due to the reason indicated above) remains a matter to discuss with the operating system designer.

**Serial Bus.** Three lines of the current P896 proposals are reserved for a serial bus. These three lines are: signal, clock and ground for the previous two. The rationale seems to be that since it is impossible to fit all the signals individually on the backplane a serial scheme is used to send messages to individual cards and use a PCR type of register to receive the data. It is almost like having a single ended packet-switching message scheme. Commands such as interrupts will be placed on this bus, also it can carry information to help in recovering from system failure on the bus.

Before looking on the black side there are reasons for the serial bus -

i. It provides a back-up to the parallel arbitration scheme when that has broken. This is useful for recovery.

ii. A great deal of information can be passed down the single serial line, that would require too many lines on the parallel bus.

However there are a variety of reasons that I think condemn it. For example:

i. The circuitry on each card is complex. At the last count it would take about 48 MSI integrated circuits to make the bus and its protocols. In defence of this it would be hoped that some manufacturers would make a custom device to control the serial bus. Although it will be a long time before this device was available.

ii. It is considerably slower than the parallel approach. When interrupts are required to initiate some action when doing recovery, these interrupts usually require immediate action in most recovery operations in order to contain errors. Having to wait while the bus is acquired and then data is put over the serial bus is totally unacceptable for any serious work on recovery. In fairness it must be noted that the parallel bus suffers from the same deficiency of not being unable to access the bus when the master wants to, but there is not the delay inherent in transmitting serial data.

iii. The necessity of having a clock with the serial bus means that there is a potential source of electrical noise on the bus that is avoidable; and it does not easily allow future expansion when the bus is required to go faster. There has to be a master clock somewhere in the system and this clock has to be fed out to all the local processors in a multiprocessor, which gives rise to an unexpected opportunity for the designer to exercise his prowess in dealing with matters such as clock skew etc. Also the master clock will be allow single point failure.

There have been suggestions that the serial bus should be made a optional feature of the level 1 or that it should only appear on level 2. The first suggestion is ridiculous - why not let everything on the P896 be optional and people do what they want. If it appears on the bus it <u>must</u> be obligatory and rigorously defined. The second suggestion is more sensible and I suspect that is what the final P896 proposal may have.

## 6.2 MECHANICAL CONSIDERATIONS

As a final comment on busses in this section mechanical matters must be considered. By its very nature the bus is very susceptible to failures due to mechanical connection. It is important that the correct choice of connector be made since the failure here will make a nonsense of the best designed system. The choices are basically direct on-card connection or a card mounted connector. The direct connector is more simple but if the cards are used in extreme environments or are being removed on a regular basis it is possible to get dirt and oxide layers on the connections. If gold is used on the edge fingers on the card it is possible that the gold-plating can dissociate from the copper, causing a high impedance layer between the copper circuit tracks and the gold. Also if the circuits are removed, care must be taken that the edge connectors are not handled or else grease layers can provide a high impedance layer.

Indirect connectors such as the DIN 41612 Euro connectors are mounted on the card by means of bolts and soldered connections. Connections between each part of the plug and socket is usually by round gold-plated pins. These pins are mounted in a protective shell so that it is impossible to touch the pins or sockets on either part, unless one has matchstick fingers. It is impossible to insert the card so that the signal lines become crossed (a singular disadvantage for the S100 bus where the power connections are adjacent). The indirect connectors are made in a variety of sizes and pin configurations. The current trend seems to be to use a 96 way shell with only 64 pins in the shell. The present DISCUS uses this connector and has proved to be completely trouble-free. Another advantage is that indirect connectors have the possibility of having more than 2 rows of connectors onto the back plane.

DISCUS uses a bus that is distributed over two 64/96 way connectors. In retrospect having the bus over 2 connectors is wasteful and if possible I would rationalise it to fit everything on one. This has the advantage of allowing bus duplication on a double Eurocard. However it is essential that discipline is used when choosing the allocation of the bus so that it becomes possible to fill the connector with a comprehensive set of non-redundant signals. (This discipline seems to have been difficult to acquire or agree with the P896 bus)

The bus should be carried by some form of printed circuit motherboard with the bus connected in a regular array between the connectors. Since this bus can be 30 cm or more it has to present a defined and repeatable impedance to the drivers on the cards. This is usually achieved over long busses by having some form of earth plane between the tracks that provides a reliable earth return. This earth plane should be an extension of the power matrix on the cards. By having a pre-manufactured board, rather than back wired connections, the basic chassis of the computer can be made up relatively quickly and cheaply.

# 7. CONCLUSIONS

In conclusion then I feel that DISCUS has been successful. It is impossible at present to give any quantitative assessment of success here. The applications for which DISCUS is being used will take a little time for complete results on its performance to be derived. However there are one or two key points that we have found in the use that make DISCUS particularly important.

## 7.1 EASE OF DEVELOPMENT

With DISCUS the overall system is split up into a number of functions that are run on a number of processors. The advantage of splitting up a system into smaller parts that can be individually written and tested and then run is one that is not special to DISCUS. Any large computer with a real time software kernel is able to do this and DISCUS is no different in this respect. It gives the advantage of code that is easier to write, debug and modify. The functions are of such a size that one man can hold a function in his head. This is important in that he can understand "completely" what he is trying to achieve, and thus what the software should be doing. The software can be tested by simulating the inputs and outputs to the function he has written, and testing the function under as many conditions as possible. When the system is brought together, provided that each function has been tested correctly, the complete system has a good chance of working with very little debugging - see more on this below.

This is all very well but what extra does DISCUS give us? The key word is isolation. At the risk of repeating myself, when a DISCUS system is designed the system is split up into a set of functions that will eventually be run on one of several separate DISCUS processors. A normal method of working would be to give each function to a different designer (assuming there are enough designers). There is some initial design (up to several months possibly) on how the functions can communicate together. This requires that the global arrays and channels between the functions are rigorously defined before any work on each function is carried out. This process will be flexible and there must be several times in the design when the channels/global store allocations are changed. However this ought not to occur too late in the development.

Some of these processors will require applications hardware on them. This will entail the design of special interface cards that live on the local DISCUS processor. If this was required on a large single processor computer, the design engineer would have two basic choices. Firstly he could take over the machine for as long as he could and develop the hardware directly on the bus or interface ports provided. (It is to be hoped that a little thought be applied to the design before he launches himself at the computer). The other method would be to make a test box that simulated the computer and then test his hardware using this. A big problem here is that he cannot be sure that this test box is true reflection of the computer. He may well have to start all over again when the applications hardware is connected to the computer.

Since DISCUS is a group of independent asynchronous computers, a single DISCUS processor can be given to the hardware designer and he can design his hardware using this. I wrote the DISCUS Monitor (see Appendix [29]) in order that designers can use a single DISCUS processor alone. In this way hardware and software design can proceed in parallel. When the processor hardware has been fully done and tested the function can be tested by using only one other DISCUS processor that is pretending to be the rest of the system. In this way a special test harness does not have to be written for the function, as one might have to do on a single processor machine. The function that is tested by the simulator function contains exactly the same software as that that will run with other real functions. As far as the function under test is concerned it has the rest of the real system around it. This concept has been particularly successful.

## 7.2 ERROR CONTAINMENT

Because function isolation is forced by the hardware it is possible to contain errors in most cases to only one processor. The processor that has faulted can take some form of recovery by itself, with no reference to any other processor. All the other processors that have not faulted can carry on as normal giving minimal disruption to the user.

## 7.3 FUNCTION DUPLICATION

Another advantage is one that is impossible with a single processor system. A multiprocessor architecture can use the principle of "incremental" computing power. In a single processor system, if it is found that the system has not got sufficient capacity to do the job (due to speed or memory space), there are virtually no simple ways to get around the problem. The ways open to the designer are: redesign the system for a faster/bigger computer, duplicate the processor (and thus rewrite the software to cope with this), or optimise the code. The latter thought often encourages the designer to rewrite the system in assembly language, not advisable. Whatever means he chose it would be open to a great deal of difficulty and expense with no guarantee of achieving what the designer intended.

However with DISCUS it is possible to duplicate functions. This means that there is another processor loaded with identical software as the one that is unable to perform its loaded function. No additional software need be written. The only difference between the processors is the hardware identity switch on the Bus Supervisor Card - hardly a great overhead. For instance, referring to Fig 10 in Section 2, the function $F_6$ provides a point through which all the data flow passes. This function is required to do too much local processing would be an ideal candidate for duplication. There are snags though, function duplication is not an inherent feature of a system such as DISCUS, the system has to be designed with duplication in mind. There are definite rules that must be applied to this task. I am not going discuss these here in detail but commend you to the main Operating System report by M.P. Griffiths [8] for more information.

Finally before looking at function splitting it worth bringing to light a snag with duplication. It is not easily possible to duplicate a function that has applications hardware (device handlers) on them. The hardware would have to be duplicated. This is not a very easy process.

## 7.4 FUNCTION SPLITTING

There is one subject that I have studiously ignored throughout the preceding sections: **automatic function splitting.** This is an extremely difficult subject and one that cannot be discussed in a few words. At no time time was it ever intended that we should look at this. However a few pointers here as to why I have ignored it may help.

In order to split a task into a set of functions we have to be able to do several things. Firstly we have to be able to describe the problem in a form that the machine can understand. Secondly we have to provide some form of guide to where devices are (this could be part of the preceding description). Lastly we have to give the machine some criteria to evaluate the "goodness" of the split.

If we start with the first two points, what we would have to develop is a way of describing our system in a fair amount of detail. Thus what we would have to do is write a description of the same order of complexity as our final program. If we could provide a suitable description for the machine to split the task correctly, it would also be able to write the program for us. In effect the language used to describe the task has become a very high level programming language. So perhaps research into this rather than function splitting is required first.

As to the goodness of the split, there would have to be a description of the parameters of this that the machine can use - very difficult.

What I can give, however, is a brief guide as to how a programmer can use a simple set of rules to split his system into functions.

i. **Maximise local store processing.** When accessing global objects there are two delays added onto the access that would not occur with accessing a local object.

   a. The arbiters will have to acquire global store. This depends primarily on whether any of the other processors are accessing global store with the "locked" attribute. In general this only occurs when a global object is being booked for access, when the software semaphores are being set. Therefore it is important that global objects are not continually opened and closed, as this will use a lot of locked accesses to global store. (This is not the only reason that this should be avoided, it is important to aid recovery.). In general I have found that the global store accesses are minimal compared to other delays.

   b. The operating system has to check the validity of the access. This is a more serious delay in the system than the hardware delays, and it is important that it should be reduced to a minimum. When splitting the system the areas of minimum data flow between functions should be the "cleavage" point when the system is being split.

   A further reason for using local store is that it aids the prevention of errors affecting the rest of the system. Providing that the errors occur when the local data is being used, it is unlikely that the global data will be corrupted.

ii. **"Brain-sized package."** Each function should be capable of being understood by one man. On the basis that smaller programs are easier to write, debug, maintain and document than large ones, the general rule is that it is better to use two smaller functions than one larger one. The only complication here is that in a system such as DISCUS, one more function means one more processor. Luckily with DISCUS this is a fairly simple (and cheap) process.

iii. **Function duplication.** Because it is desirable to replicate functions for throughput and redundancy, it is important that the system be split to take account of function duplication. Exactly how this should be done I have left to the report on the operating system [8] to go into in greater detail. However it does bring to light a very interesting point - how should the hardware interfaces be done on DISCUS? As I have indicated above duplicating (or more) the software only requires another processor; if this processor is only looking at global store, then this is easy. However if this processor has some external applications hardware on it then it is more complicated. Replicating hardware is more difficult, requiring specialised connections to the incoming data. Since we cannot duplicate the hardware interface processor, it important that this processor only gets the incoming data into some digestible form (conditions it) and passes it on to the other functions in the system which can be duplicated. Since the processor is only carrying out a very simple input/output function it is unlikely that any "bottlenecks" will occur there.

## 7.5 DISADVANTAGES OF DISCUS

What we have been primarily concerned with in DISCUS is recovery in the software. Redundancy of hardware has not been considered and there are many places in DISCUS that single point failure can occur. In general if one component of DISCUS fails then the whole fails, for instance the global store and its accessing. The functions could be duplicated - however there are traps for the unwary here also. It is possible that a processor can fail and thus it is possible for it to leave a selection of global objects to be left in an open state and thus prevent any other processor from accessing them. This will cause the system to crash with only operator interference to restart the system. It would be possible for the booking mechanisms at the operating system level to be corrupted and a channel/array booked for a nonexistant processor. Why not, you may ask, have another processor detect this condition and take some remedial action to remove the offending processor?

This would require either a single master processor, or the other processors being able to agree on when another processor has failed and then throw that one off. At present the former is impossible as all the processors have equal priority and under <u>no</u> circumstances can one pre-empt another.

The latter would require each processor being able to decide when a processor was wrong, and this implies that it must have some knowledge of what that one was trying to do. At present DISCUS processors do not have any concept of what their fellows are trying to do, in fact it could be said they have no knowledge of whether they have any fellow processors at all. As far as each processor is concerned they get data from an external device or global store location, and send it back to similar devices. How it got there, whether by another processor accessing global store, by a specialised device on the global store or even black magic, is of no concern.

Finally I am not at all happy with the present DISCUS bus. Experience and actually trying to use it for a variety of things has convinced me that it is pretty awful. For instance there is a high speed clock adjacent to the local bus acknowledge line, the latter has to be treated with care if noise pick-up is to be avoided. Also the DMA request/grant lines are parallel to the data lines. This causes induced noise from the data bus line drivers turning on. It would definitely not be used for the next machine.

## 7.6 SUCCESS (OR OTHERWISE) OF THE PRESENT SYSTEM.

There is no doubt that the present system has been successful. Several single DISCUS processors without global store have been used, both as development aids for producing software and hardware in the multiprocessor, and for people who required a simple micro-based system. Four complete DISCUS computers have been made and all work most reliably. One of them has been delivered to someone who had nothing to do with the design of the system, but merely wanted a multiprocessor for a task. He has used it with no problem for over two years. The application that we have is a small local telephone exchange. At the time of writing it has been running or about 18 months. The application has demonstrated that a fairly large system with 6 functions can be written with the minimum of effort. Each of the functions was tested before the complete system was integrated in accordance with the ideas above. The actual integration only took an afternoon with only a few faults that were easily rectified. Function duplication was demonstrated by having four processors run identical software.

Another disadvantage of DISCUS is speed. DISCUS runs relatively slowly. Both these applications indicated to us that the main speed limitation lies in the operating system. The basic cycle time of the processor is about 2.5 microseconds. This time is for a simple local store access. The global store accesses take anywhere between this time and about 15 microseconds. This time will increase to several milliseconds if there are many DISCUS processors accessing the global store in Memory-Locked mode and accessing large data arrays. In normal working we have found that the majority of the time is spent by the operating system doing the basic housekeeping when accessing the global objects. Also the operating system is written in a high level language which in our case does not have a very efficient compiler available to us. To quote ref [8] - "Whether this is due to the fact that the operating system is written entirely in CORAL and may simply need "tuning" to speed it up, or indicates a more fundamental flaw in that the (operating system) protocols are inefficient, has yet to be determined."

## 7.7 THE FUTURE

There is the new Mark 1.5 DISCUS that I have discussed briefly in the other sections that we can use for interim experiments in recovery. However parallel hardware research should be done to look at some new architecture based on DISCUS. I have looked at some areas of this in some detail in the other sections, so I will not go over this again. Much has to be done on recovery using DISCUS, or more advanced machine, at present we have given ourselves the hardware and operating system to provide the facilities; how we use them to provide distributed recovery is yet to be done.

So as a final memory jogger the following is a list of areas that would be useful to explore with a new DISCUS.

i. Looking at memory management using the new manufacturer supplied devices (such as the Zilog Z8010). At present DISCUS has no form of memory management - it is all absolute addressed, and facilities such as bounds checking on data areas could be used. The speed of the system would be made much faster since what is done in software at present could be done in hardware. The MMU could be used as the basis of capability-like structure to give DISCUS a more secure computing base.

ii. Using a fast 16-bit microprocessor to help increase the overall system speed over the present DISCUS.

iii. Using several global busses to provide redundancy, and to provide alternative paths to global store for reducing contention.

iv. Look at some form of data protection on the bus (parity etc).

v. Investigate processor redundancy which, as I said in Section 1, is something that we never considered.

## REFERENCES

These references represent only a small fraction of the total available on multiprocessors. The reference marked with an asterisk contains a useful list of further references.

1. CASAGLIA, G.F.
"Distributed computing systems: a biased review"
Euromicro Newsletter 4, No 2, p 5, 1976.

2. SEARLE, B.C., et al
"Tutorial Microprocessor applications in multiple processor systems"
Computer, Oct 1975, p 11.

3. BAKER, K.D.
"Functional decomposition on multi-microprocessor systems"
The Radio and Electronic Engineer
Vol 47, No 11, pp 497-504, Nov 1977.

4. Intel iSBC80-30 Single Board Computer Hardware Reference Manual
No. 9800611

5. Intel MULTIBUS Definition

6. FRASER, A.G.
"Loops for data communications"
Bell Laboratories Computing Science Tech Report 24, Dec 1976.

7. TONG, H.D.
"Microprocessor based multiprocessor ring structured network"
AFIPS NCC 1975, 567.

8. GRIFFITHS, M.P.
"The DISCUS Operating System"
RSRE Report No. 82009

9. CHANEY, T.J. and MOLNAR, C.E.
"Anomalous behaviour of synchroniser and arbiter circuits"
IEEE-TC (Coress), Vol C22, No 4, April 1973, pp 421-422.

10. PECHOUCEK, M.
"Anomalous response times of input synchronisers"
IEE-TC, Vol C25, No 2, Feb 1976, pp 133-139.

11. Z8000 processor and Z8010 Memory Management Unit Data Sheets
Zilog Corporation.

12. Intel 8080/88085 Assembly Language Programming Manual, 9800301

13. NICHOLS, H.K. and FIELD-RICHARDS, H.S.
"DISCUS - A distributed control microprocessor system"
Microprocessor and Microsystems journal,
Vol 3 No 6 July/Aug 1979, pp 267-270

14.* SATYANARAYANAN, M.
"Commercial Multiprocessing Systems"
IEEE Computer, May 1980, pp 75-116.

15. PLUMMER, W.W.
"Asynchronous Arbiters"
IEEE Trans on computers, Vol c-21, No 1 Jan 1972, pp 37-42

16. CORSINI, P.
"Self-synchronising Asynchronous Arbiter"
Digital Processes, 1 (1975), pp 67-73.

17. COURVOISIER, M.
"A programmable arbiter for multiprocessor systems"
Digital Processes, 5 (1979), pp 271-281

18. GRIFFITHS, M.P.
"Input/Output Package for use with the GEC RCC80 CORAL compiler"
RSRE Internal document (not published)

19. SHRIVASTAVA, S.K.
"Stucturing Distributed Systems for Recoverability and Crash Resistance"
University of Newcastle-on-Tyne Memo SRM/227.

20. JOHANSSON, L-A.
"Virtual Memory Management for Microcomputers in Real Time Applications"
Euromicro Journal, 5 (1979), pp 235-238.

21. BISHOP, P.G.
"The Design of Software for Distributed Control Systems"
Euromicro Journal, 6 (1980), pp 135-143.

22. HERRING, J.
"The Intel 8086, Zilog Z8000 and Motorola MC68000 *Microprocessors*"
Euromicro Journal, 6 (1980), pp 135-143.

23. SAMI, M.
"Reliability and Self-diagnosis Aspects of microprocessor controlled Instrumentation Systems"
Euromicro Journal, 6 (1980), pp 343-345.

24. *Intel ISIS-11 Operating System Manual, 9800306D*

25. Intel Applications note on Hex format.

26. Intel 8080 Microcomputer Systems User's Manual, 9800153
Intel 8085 Microcomputer Systems User's Manual, 9800366

27. "IEEE 896, A proposed Standard Backplane Bus Specification for Advanced Microcomputer Systems"
Draft 4.1, Revised January 1st, 1982.

28. LEBLING, P.D. et al
"ZORK: a Computerised Fantasy Simulation Game"
IEEE Computer, April 1979, pp 51-59.

29. H.S. FIELD-RICHARDS,
"The DISCUS Hardware Description and Appendices"
RSRE Memo No. 3483

30. Intel MULTIBUS Interfacing, AP 49

## GLOSSARY OF TERMS AND SYMBOLS USED IN DISCUS

Note: The following terms are presented, not as points for discussion, but as precise definitions on how they will be used throughout this report.

### FUNCTION

A function is a closed piece of code the execution of which would carry out a particular job in the system. A "closed" piece of code means that there is no path through the function which involves a transfer of control to code outside of the function except to the supervisor.

### PROCESS

A process is the execution of a function for a given set of parameters.

### TRANSACTION

A transaction is a set of transient data which causes a process on a given function to run.

### CPU

A cpu is the device that interprets and performs given machine code instructions. A microprocessor can be a cpu.

### PROCESSOR

A processor consists of a cpu and memory, and the means to connect them to run programs.

### COMPUTER

A computer is a complete usable installation that implies processor(s), and peripherals, such as VDU and disks.

## GLOSSARY (continued)

| | |
|---|---|
| AAR | - Auxilliary Address Register |
| ADRx | - Main address bus |
| CARD | - One double Eurocard |
| CPU | - Central Processor Unit (see definition above) |
| CRATE | - One 19 inch wide assembly of CARDS |
| DATx | - Main data bus |
| DISCUS | - Distributed Control Microprocessor System |
| GLOBAL CRATE | - The global store CRATE |
| GRNTS | - Grant Store |
| LOCAL CRATE | - A set of local processors in a CRATE |
| PROM | - Programmable ROM |
| RACK | - A vertical assembly of CRATES |
| RAM | - Random Access Memory* |
| REQS | - Request Store |
| ROM | - Read Only Memory |
| STGR | - Grant store |
| STRQ | - Request store |
| VDU | - Visual display unit |

## IMPORTANT NOTES

1. Where a signal name is followed by an '/', it signifies that this signal is negative true, i.e. for TTL logic the 0v is a logical one and +5v is a logical zero.

2. The "k" and "M" prefixes to the terms bytes, bits etc, refer to the binary equivalent NOT the decimal multipliers, for instance 1k bytes=1024 bytes not 1000 bytes.

* Random Access Memory in this case means read/write memory. In the true sense of the word ALL the store in DISCUS is random access, since it can all be accessed with equal ease.

## ACKNOWLEDGEMENTS

Firstly I would like to thank **Howard Nichols** whose efforts got DISCUS started and was the source of the basic software architecture. Howard wrote the operating system and the initial system generators for DISCUS. He provided much discussion on the hardware architecture of DISCUS.

When Howard left **Mike Griffiths** took over the work and produced the DISCUS report on the operating system. Also the work on recovery was done by him, and I would like to thank him for all his advice and proof reading, especially his patience when I was dense and/or bothersome.

DISCUS has been a co-operative venture so a sincere thank you to all those concerned who have used or been involved with DISCUS. These are **Ian Outram**, **Terry Connelly**, **Steve Bracking**, and **Chris Turner**.

I would like to thank **Jeannie Kilby**, **Deirdre Lindup** and **Lesley Kinder** for doing the initial typing on the word-processor for me.

Finally a blanket thank you to all who answered queries and proffered advice, wanted or unwanted.

## NOTES ON THIS REPORT

This report was originally submitted as a Ph.D. disertation with Imperial College and represents only a small part of the total report on the DISCUS multiprocessor. There are two more reports/memos concerned with DISCUS which describe it in more detail. These are "The DISCUS Hardware Description and Appendices" [29] and "The DISCUS Operating System" [8]. Reference is made to them throughout this document. It is impossible to read this report without refering to them.